Kamiwaza AI Platform

Offline Installation Guide

VERSION 1.3.4

Table of Contents

1. Getting started **5**
- 1.1 Introduction 5
- 1.2 Quickstart 9

2. Release notes **14**

3. Installation **24**
- 3.1 Installation Process 24
- 3.2 System Requirements 25
- 3.3 Licensing 40
- 3.4 Install Airgapped 44
- 3.5 Gpu Nodes 55
- 3.6 Upgrade 81

4. Environments **94**
- 4.1 Environments 94

5. Kubernetes with Istio **95**
- 5.1 Prerequisites 95
- 5.2 Install 121
- 5.3 Uninstall 130

6. OpenShift with GreyMatter **132**
- 6.1 Prerequisites 132
- 6.2 Install 149
- 6.3 Uninstall 160

7. Configuration **163**
- 7.1 Configuration 163
- 7.2 Database Schema Lifecycle 168
- 7.3 Routing Modes 171
- 7.4 Workroom Storage S3 176

8. Workrooms **184**
- 8.1 Runtime Contract 184

9. Security **187**
- 9.1 Admin Guide 187
- 9.2 Consent And Classification 193
- 9.3 Rebac Overview 195
- 9.4 Rebac Deployment Guide 198
- 9.5 Rebac Validation Checklist 202

10. Federation **206**
- 10.1 Overview 206
- 10.2 Identity Trust Modes 210
- 10.3 Setup 214
- 10.4 Retrieval 222
- 10.5 Job Submission 227
- 10.6 Operations 234
- 10.7 Execution Gates 244

10.8 Gate Packages 252
10.9 Api Reference 257
10.10 Release Notes 1 3 267

11. Models 270
11.1 Overview 270
11.2 Novice Mode 274
11.3 Gui Walkthrough 276
11.4 Downloading Models 283
11.5 Deployment 284
11.6 Tenant Mode Inference 287
11.7 Placement Overview 313
11.8 Placement Hardware Classes 316
11.9 Placement Fractional Serving 319
11.10 Placement Deployment Guide 323
11.11 Bedrock 328
11.12 Aws Transcribe 337
11.13 Openai Compatible Chat 346
11.14 Openai Compatible Transcribe 353
11.15 Troubleshooting 356
11.16 Release Notes 1 3 358

12. Extensions 362
12.1 App Garden 362
12.2 Tool Garden 365

13. Distributed Data Engine 367
13.1 Overview 367
13.2 Data Connectors 371
13.3 Data Catalog 378
13.4 Retrieval Service 382

14. Observability 384
14.1 Observability 384

15. Architecture 389
15.1 Platform Overview 389
15.2 Components 391
15.3 Core Services 393

16. Use Cases 396
16.1 Use Cases 396
16.2 Building A Rag Pipeline 398

17. Help & Fixes 410
17.1 Help & Fixes 410

18. Runbooks 414
18.1 Spicedb Maintenance Upgrade 414
18.2 Ontology Graph Viewer 418

19. SDK 422
19.1 SDK - Introduction 422
19.2 SDK - REST API Reference 423

20. Python SDK Services **424**
20.1 SDK - Activity 424
20.2 SDK - Apps 427
20.3 SDK - Auth 433
20.4 SDK - Authz 436
20.5 SDK - Catalog 437
20.6 SDK - Cluster 440
20.7 SDK - Connectors 444
20.8 SDK - Context 450
20.9 SDK - Embedding 458
20.10 SDK - Enclaves 463
20.11 SDK - Extensions 464
20.12 SDK - Federations 472
20.13 SDK - Ingestion 476
20.14 SDK - Jobs 478
20.15 SDK - Lab 480
20.16 SDK - Markings 483
20.17 SDK - Models 484
20.18 SDK - Openai 488
20.19 SDK - Retrieval 490
20.20 SDK - Serving 494
20.21 SDK - Skills 498
20.22 SDK - Tools 503
20.23 SDK - Vectordb 510

21. Extensions **513**
21.1 Kamiwaza Extensions 513
21.2 Developer Guide 514
21.3 Compatible Catalogs 543
21.4 Selection Qualification 548

22. OmniParse **550**
22.1 Service Guide 550

23. Kaizen **552**
23.1 User Guide 552
23.2 Changelog 570

24. Skills Library **574**
24.1 User Guide 574

25. Graphiti **577**
25.1 Service Guide 577
25.2 Changelog 581

26. Workroom Manager (Deprecated) **582**
26.1 User Guide (Deprecated) 582

Welcome to Kamiwaza Docs

Welcome to the official documentation for Kamiwaza, the solution for building, deploying, and scaling production-grade AI applications.

Kamiwaza Overview

Kamiwaza is a comprehensive AI orchestration platform that provides developers and IT teams with the tools to manage the entire lifecycle of their AI models and applications. From data ingestion and vectorization to model serving and monitoring, Kamiwaza simplifies the complexity of the modern AI stack, allowing you to focus on building innovative features, not wrestling with infrastructure.

Getting Started

Ready to dive in? The fastest way to get started is to follow the **Installation Guide** (see online documentation) to set up your environment.

Looking for Inspiration?

Check out the **Use Cases** (see online documentation) section to see practical implementations and get inspired by what's possible with Kamiwaza.

Explore the Documentation

⚡ Quickstart

Validate your platform, storage, security, and key user workflows before rollout.

Open the Quickstart → (see online documentation)

🚀 Installation Guide

Review supported installation paths, prerequisites, and system requirements.

Start the Installation → (see online documentation)

🔒 Security

Configure authentication, authorization, and access control for your platform.

Security Guide → (see online documentation)

🤖 Models

Learn how to deploy, manage, and serve AI models on the platform.

Manage Models → (see online documentation)

🪴 App Garden

Explore a curated marketplace of pre-built AI applications and services.

Explore Apps → (see online documentation)

💾 Distributed Data Engine

Understand the data layer, from relational data in CockroachDB to vectors in Milvus.

Learn About Data → (see online documentation)

📊 Observability

Monitor platform health, track metrics, and debug issues with built-in observability.

View Observability → (see online documentation)

🏛️ Platform Architecture

Get a high-level overview of the system's components and design philosophy.

View the Architecture → (see online documentation)

💡 Use Cases

Learn how to build real-world AI applications with step-by-step guides and practical examples.

Explore Use Cases → (see online documentation)

📦 SDK

Interact with the Kamiwaza platform programmatically using the Kamiwaza Python SDK.

Explore the SDK → (see online documentation)

Need Help?

If you have questions or run into issues, use any of these channels:

- Join the Discord community (https://discord.gg/cVGBS5rD2U)
- Visit the Kamiwaza website (https://www.kamiwaza.ai/)
- Visit the GitHub organization (https://github.com/kamiwaza-ai)
- Try the client SDK (https://pypi.org/project/kamiwaza-sdk/)
- Contact the support team (https://portal.kamiwaza.ai/_hcms/mem/login?redirect_url=https%3A%2F%2Fportal.kamiwaza.ai%2Ftickets-view)

Kamiwaza is committed to making your experience as smooth as possible.

Quickstart

Use this guide after installing Kamiwaza. It is intended for single-user deployments and administrators who are preparing for a broader rollout.

If you are still preparing the environment, start with:

- System Requirements (see online documentation)
- Installing Kamiwaza (see online documentation)

Before You Start

Make sure you have:

- a completed Kubernetes deployment of Kamiwaza
- cluster or environment admin access appropriate for your deployment
- one administrator account and one standard user account for validation
- enough CPU, RAM, GPU, and disk for the models and extensions you plan to validate
- a plan for object storage if users will upload files or use the Skills Library
- network access to fetch models, or staged artifacts if your environment is air-gapped

This guide assumes the cluster is already installed. Kamiwaza 1.3 runs on Kubernetes, with Keycloak for sign-in.

Use System Requirements (see online documentation) to confirm hardware sizing, storage expectations, and platform prerequisites before installation. Use Installing Kamiwaza (see online documentation) for the supported installation workflow for your platform.

1. Confirm the Kubernetes Deployment Is Healthy

After deployment, confirm the core workloads are running:

```
kubectl get pods -n kamiwaza
```

Then confirm you can reach your environment entry points. Common endpoints include:

- Frontend: your Kamiwaza web URL
- API documentation: your Kamiwaza API docs URL
- Ray Dashboard: if exposed in your environment

Use the gateway URL and hostnames provided for your deployment.

If the UI does not load cleanly or core pods are not healthy, pause here and review Help & Fixes (see online documentation) and Observability (see online documentation) before moving on.

2. Validate Administrator and End-User Access

Use at least:

- one administrator account for platform setup and extension deployment
- one standard user account that reflects everyday end-user behavior

Create or confirm these accounts through the user-management flow available in your deployment.

Make sure both accounts can sign in through Keycloak or the identity provider you configured.

For user lifecycle details, role definitions, and Keycloak administration, see the Administrator Guide (see online documentation).

3. Validate Authentication and Session Defaults

Before inviting broader users, confirm the environment matches your security posture:

- Keycloak and any configured OIDC, SAML, or LDAP integration behave as expected
- consent, banner, and handling-marking requirements are configured as intended
- administrator access to deployment and audit logs matches your environment requirements
- App Garden session defaults, including ephemeral-session policy, match your environment

Use these references while validating the environment:

- Administrator Guide (see online documentation)
- Consent and Classification (see online documentation)
- Observability (see online documentation)

Use the Administrator Guide (see online documentation) for detailed authentication setup, external identity integration such as LDAP or Keycloak, and secrets-management guidance. Use Observability (see online documentation) for log configuration and validation of your deployment logging path.

If you are validating a federal deployment that uses CAC, use the CAC Overview (see online documentation) alongside this quickstart.

4. Configure Workroom Storage

If users will upload files, preserve workroom context, or use the Skills Library, object storage must be available before release validation. Without it, common extension workflows fail or behave inconsistently.

Default self-managed installs use namespace-local SeaweedFS for workroom object storage, so no object-store configuration is needed here. Its volume uses the block storage class you name at install; see Block storage in your environment's prerequisites: Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation). Follow AWS S3 Workroom Storage (see online documentation) only when workroom content must live in external AWS S3.

When this step is complete, validate that:

- the bucket and region are configured correctly
- credentials are available to core services
- uploads and Skills Library package operations no longer fail with storage errors

5. Verify Observability and Deployment Logs

Before you validate extensions, make sure administrators can see enough telemetry to diagnose failures quickly.

Check:

- Kubernetes pod health with `kubectl get pods -n kamiwaza`
- Kubernetes logs with `kubectl logs -n kamiwaza <pod>`
- the UI log viewer for deployment logs
- any OpenTelemetry or external log-forwarding integration configured for your environment

Use the Observability (see online documentation) guide to verify logger-service coverage and cluster logging expectations.

6. Deploy a Validation Model

Use a small, known-good model that matches your hardware profile and is easy to redeploy during validation.

Recommended validation flow:

1. Open **Models** in the Kamiwaza UI.
2. Search for a model appropriate for your hardware profile.
3. Download the exact artifact you plan to validate.
4. Deploy the model and wait for the deployment to report healthy status.
5. Record the model name, engine, and deployment details in your release notes or validation log.

If you need help choosing or deploying a model, use:

- Models Overview (see online documentation)
- GUI Walkthrough (see online documentation)
- Deployment (see online documentation)

7. Validate an End-User Workflow

Once the platform is up, validate at least one end-to-end end-user path instead of stopping at infrastructure checks.

Validate Kaizen

1. Deploy **Kaizen** from **App Garden**. Choose **Kaizen**, not **Kaizen (Legacy)**.
2. In Kaizen, open the user menu → **Admin area** → **Models**, choose a **Default chat model**, and save. A locally served chat model needs a context window of at least 32,768 tokens.
3. Run a search, then start a chat that uses a document or a tool relevant to your environment, and confirm the answer cites its sources.

Reference: Kaizen documentation (see online documentation), including Install & upgrade (see online documentation) and Your first search & chat (see online documentation)

Validate Workrooms

> **Version note:** This version supports launching Kaizen inside a workroom context and enforcing basic role-aware writes, but the full non-global shared-workroom Kaizen contract described elsewhere in the docs may depend on a later patch or feature build. Validate the exact behavior present in your deployed Kaizen build before treating the checks below as release gates.

1. Open **Workrooms** from the navigation sidebar. The page it opens is titled **Workroom Manager**; it is part of Kamiwaza, not the deprecated extension. Workrooms are built into Kamiwaza; on an installation that shows **Workrooms** in the sidebar, do not deploy the deprecated Workroom Manager extension from App Garden. If **Workrooms** is not in the sidebar, workrooms are either turned off or provided through the extension; ask your administrator, and for extension installations see the Workroom Manager User Guide (Deprecated) (see online documentation). If workrooms are turned off, skip the rest of this section.
2. Create a workroom with a title, classification banner, and optional labels.
3. Decide whether the workroom should be persistent or temporary.
4. Launch Kaizen from inside the workroom and confirm the app opens successfully.
5. If sharing is enabled in your environment, add at least one Contributor and one Viewer.
6. If your deployed Kaizen build includes shared non-global workroom conversations, verify the Contributor can see the same workroom agents and conversation history, add a later turn, and retain acting-user attribution on that turn.
7. If your deployed Kaizen build includes shared non-global workroom conversations, verify the Viewer can see the same shared history but cannot create, send, upload, or edit.
8. Verify the Global Workroom still preserves personal Kaizen visibility instead of cross-user sharing.

Reference: Workroom Runtime Contract (see online documentation)

8. Capture Readiness Evidence

Before calling the environment ready, record what was actually validated.

Suggested checklist:

- administrator sign-in path confirmed

- standard-user sign-in path confirmed
- object storage verified for workroom and Skills Library workflows
- at least one model deployed successfully
- Kaizen deployed and exercised successfully
- Workrooms exercised successfully (where workrooms are enabled)
- deployment logs and cluster health reviewed for unexpected errors

Next Steps

After this quickstart, most teams move into these guides:

- Administrator Guide (see online documentation)
- AWS S3 Workroom Storage (see online documentation)
- Observability (see online documentation)
- Help & Fixes (see online documentation)

Kamiwaza 1.3 release notes

Kamiwaza 1.3 installs with Helm into a Kubernetes cluster that you run. This page covers what is new in 1.3.0, what changed from 1.2, what is deprecated, the known issues for the 1.3 line, and each patch release, newest first. The latest release is 1.3.4 (see online documentation).

- Supported platforms and configurations: System Requirements (see online documentation)
- Installing: Installing Kamiwaza (see online documentation)
- Upgrading within 1.3: Upgrade Kamiwaza (see online documentation)

New in 1.3.0

1.3.0 was delivered as `1.3.0-rc.1`. That chart and its release manifest are the 1.3.0 build.

Installation and distribution

- **Helm install into a cluster you run.** Kamiwaza installs with one `helm upgrade --install` into a single namespace, as a namespace-scoped identity, without cluster-admin. Kamiwaza no longer sets up Kubernetes, the service mesh, storage, or GPU drivers. A platform administrator prepares those, following the prerequisites page for each environment.
- **Two documented environments:** Kubernetes with Istio, and OpenShift with GreyMatter. Each has prerequisites, install, and uninstall pages. On OpenShift, `global.platform: osgm` selects the OpenShift and GreyMatter settings. See Installation Environments (see online documentation).
- **Public distribution.** The chart and every image are published at `oci://ghcr.io/kamiwaza-ai/releases/kamiwaza` and pull without credentials. Each release publishes a manifest that lists every image.
- **Air-gapped installation from your own registry.** Copy the release into your registry and set one registry-prefix value; it re-addresses every image, including App Garden extensions and inference engines. See Install Without Internet Access (see online documentation).
- **Images pinned by digest,** each with its third-party license notices. Images built for a release are published under the version's tag, such as `release-1.3.3`; an image with no tag of its own gets `release-1.3.3-` followed by 12 characters of its digest. Images that carry their own version, such as PostgreSQL, and App Garden extension images keep their own tags. A published tag never moves.
- **Signed license file.** Every installation needs a signed `license.lic`, checked at startup without contacting Kamiwaza. API responses report the license state, and the UI shows a banner as the term ends. An expired term does not stop the platform. Accepting the EULA is a required install value. See License File (see online documentation).
- **Documented uninstall** for each environment.

Platform

- **Istio Gateway API routing.** Kamiwaza attaches its routes to your Istio gateway from its own namespace and checks access at the gateway. Models, apps, and tools are served under paths on one HTTPS origin (`/runtime/models|apps|tools/<id>`). See Routing modes (see online documentation).
- **Envoy edge without Istio (early release).** The chart can deploy a standalone Envoy edge for clusters that do not run Istio (`global.ingress.provider: envoy`). It is not yet a documented install environment.

- **Relationship-based access control on SpiceDB (early release).** Access control is relationship-based by default, with SpiceDB as the backend.
- **Storage.** Kamiwaza uses the block StorageClass you provide. Object storage is a built-in SeaweedFS in the namespace by default, or your own S3-compatible service.
- **Generated admin password.** If you do not set one, the install generates the admin password and stores it in the `kamiwaza-user-admin` Secret.

App Garden and extensions

- **Bundled extension catalog.** Each release ships a fixed set of extension versions, so the App Garden works without reaching the online catalog.
- **Administrators choose extension versions.** An administrator selects the version of each extension that new deployments use. Running deployments are not changed.
- **Kaizen is now the new Kaizen (v4).** The `kaizen` entry is Kaizen 0.7.1, shown as Kaizen.Next in 1.2. Kaizen v3 continues as **Kaizen (Legacy)** (`kaizen-legacy`, 2.1.4), for existing deployments.
- **Extension versions in 1.3.0:** Kaizen 0.7.1, Kaizen (Legacy) 2.1.4, OmniParse 2.2.3, Skills Library 0.5.2, Milvus 2.4.11, Graphiti 2.6.7, Kamiwaza DDE 2.5.4, Data Connector Builder and Runtime 0.2.5 (alpha), Google and Microsoft 365 connectors 0.1.1, and the Hello Web sample 1.0.0.
- **Extension developers:** the SDK and `kz-ext` support Kamiwaza 1.3. Upgrade `kz-ext`, because earlier versions report Kamiwaza 1.3.0 as unsupported. `kz-ext` refuses build arguments that look like credentials; use BuildKit build secrets instead.

Models and GPUs

- **Tenant-mode inference.** A namespaced installation can serve local models on CPUs and NVIDIA GPUs without reading the cluster's nodes. The cluster owner describes the hardware in a signed profile, and Kamiwaza places each deployment only on that hardware: CPU, whole GPUs, MIG slices, or GPUs shared through the VRAM plugin. The owner tool ships in the Core image. See Tenant-mode inference (see online documentation).
- **Engine catalog.** Tenant mode serves llama.cpp, vLLM, diffusion, and whisper.cpp from digest-pinned images, and picks the newest image that both the GPU and the host driver can run.
- **Image generation on NVIDIA GPUs** with a first-party diffusion engine.
- **vLLM 0.28.0** (from 0.23.0).
- **Deployments report what they serve:** context window, and reasoning and vision support, for local and external models.
- **Claude 5 models through Amazon Bedrock.**
- **Embedding improvements,** including a bundled default embedding model for installations without internet access.
- **GPU node guide** with driver requirements and a check for each step. See GPU Nodes (see online documentation).
- **Hardware:** NVIDIA GPUs. AMD GPUs are not supported in 1.3. Platform nodes are x64/amd64. NVIDIA GB10 (DGX Spark, arm64) nodes run inference engines; vLLM on GB10 is available from 1.3.1, and diffusion from 1.3.3.

Federation

- **Delegated jobs run without KubeRay,** so namespaced installations can run them.
- **The execution gate set in Helm takes effect,** and the API reports which gate is in force.
- **Retrieval responses say when they are truncated.**
- **Clearer pairing errors,** and a private certificate authority is shared with peers during pairing.

See Federation 1.3.0 release notes (see online documentation).

Changes from 1.2

- **No in-place upgrade from 1.2 or earlier.** Contact your Kamiwaza representative to plan an assisted migration to a new installation.
- **The installer scripts are removed,** including the online installer that set up Kubernetes on the host and the offline bundle. Kamiwaza is no longer distributed through Keygen.
- **A license is required.** Kamiwaza does not start without a valid signed license file.
- **macOS is no longer supported.** See System Requirements (see online documentation).
- **Removed components:** the earlier ingress routing provider, replaced by Istio (see Routing modes (see online documentation)), Rook/Ceph storage, KubeRay, and the bundled Grafana and Loki.
- **GPU serving needs setup.** A default installation cannot see GPUs, and a GPU deployment returns `422` until the cluster owner turns on tenant-mode inference or grants Kamiwaza read access to nodes. See GPU Nodes (see online documentation).
- **The `kaizen` entry is a different product.** There is no migration from Kaizen v3 to Kaizen v4 in any release. Existing v3 deployments keep running. Adopting Kaizen v4 means a new deployment. The `kaizen-next` name is retired.
- **Workrooms are built into Kamiwaza,** and the Workroom Manager extension is retired. The Workrooms page is off by default.
- **Vespa is not offered** in the 1.3 App Garden. Existing Vespa deployments and data are kept.
- **Install and upgrade with Helm only.** Applying rendered manifests, for example through Argo CD or Flux, is not supported.
- **Model serving changes:** see Model serving 1.3.0 release notes (see online documentation).

Deprecated, removed in 1.4

- **Kaizen (Legacy)** (`kaizen-legacy`) is removed in 1.4.0, with no migration to Kaizen.
- The workroom `classification` field, replaced by `level_id` .
- `global.workroomManager.surface: extension` .
- A plain string for `global.greymatter.proxy.image` . Use the `registry` , `repository` , and `tag` fields, or `digest` .

Known issues

Open in 1.3.4:

- **A new Kaizen version does not keep data.** A new Kaizen deployment starts empty, and stopping a Kaizen deployment in the App Garden deletes its data without a warning in the dialog. See Upgrade Kamiwaza (see online documentation).
- **Kaizen starts with no embedding model selected.** Until an administrator selects one, documents are indexed for keyword search only. Selecting a model does not index documents automatically, including documents uploaded afterward; choose **Index** on each document, or **Index selected**. The same applies to a new Kaizen deployment.

- **Kaizen's Models screen says the embedding model must return 768-dimensional vectors.** The 384-dimensional MiniLM model also works.
- **The Software Engineer agent's language-server tools need internet access.**
- **Extra Python or npm packages** that the Kaizen images do not include need a package registry or an internal mirror.
- **Model endpoints over plain HTTP** or private addresses require `core.scheduler.allowInsecureModelEndpoints: true`. Turn it on only for endpoints on a network you trust. Cloud metadata addresses stay blocked even when it is on.
- **Istio ambient mode is not supported.** Istio must run in sidecar mode.
- **Tenant-mode inference** is qualified on k0s with Kubernetes 1.36. A deployment across more than one GPU is not qualified, and MLX is not available in tenant mode.
- **Demoting a linked administrator does not remove Keycloak admin access.** The account can retain administrative access through both its existing session and a new login. The fix is deferred to 1.4. Deleting and re-adding the user does not resolve this limitation. Contact Kamiwaza Support before demoting a linked administrator.
- **The external identity provider form cannot select a role claim.** The default claim is `groups`, which Google does not send. A Google provider saved in the UI denies everyone by default; **Admit every user** instead admits everyone who can sign in through that provider. Domain or email mapping requires the API with `role_claim: "hd"` or `role_claim: "email"`. Google sign-in with these mappings has not been validated on 1.3.4; contact Kamiwaza Support before using it. Registering Google as generic OIDC is not a workaround.

Fixed:

- Air-gapped installations failed on the App Garden catalog sync. Fixed in 1.3.1.
- The bundled embedding model could try to download from the internet. Fixed in 1.3.1.
- On OpenShift with GreyMatter, a model deployment could start before GreyMatter was ready and not join the mesh. Fixed in 1.3.1.
- Workroom deletes and membership changes could fail with `503` under load. Fixed in 1.3.1.
- Tenant-mode diffusion used the engine image from before the 1.3.1 base-image refresh. Fixed in 1.3.2.
- Kaizen chat failed on Envoy ingress without Istio. Fixed in 1.3.3 for Kaizen that an administrator deploys from the catalog.
- After a platform upgrade, the App Garden kept the extension versions selected before it until an administrator selected the new ones. Fixed in 1.3.4.

Patch releases

1.3.4

Security fixes, owner controls for tenant-mode inference, tool calling on tenant-mode vLLM, Kaizen 0.7.11, and Kaizen (Legacy) 2.1.8.

Before upgrading

- **Back up the Kamiwaza database.** After the upgrade, Core from 1.3.3 or earlier does not start on the upgraded database, so you cannot go back with `helm rollback`. Going back means restoring a backup taken before the upgrade. See Before you start (see online documentation).
- **Upgrade with `helm upgrade`, not by changing only the Core image.** 1.3.4 adds database migrations, which the `core-db-init` Job runs during `helm upgrade`. A Core image changed any other way, for example with `kubectl set image`, refuses to start until that Job has run for the new

image, and reports `SchemaAnchorError: The db-init service migration phase has no completion receipt for this image.` This is expected, and the database is not damaged. Rerun the upgrade command, as described in Upgrade (see online documentation). Do not stamp the database by hand or restore a backup.

- **A Global Milvus VectorDB with a name other than `milvus-global-workroom`:** from 1.3.4 only `milvus-global-workroom` is the Global Workroom's default. Check whether a custom-named instance receives Global ingest, and choose what to do, as described in A custom-named default Global Milvus (see online documentation).
- **Kubernetes with Istio and OpenShift with GreyMatter:** App Garden apps no longer receive the Kamiwaza sign-in cookies. Apps that call Kamiwaza as the signed-in user stop working until an administrator trusts them in your values file, except Kaizen and Kaizen (Legacy) that an administrator deployed from the App Garden on 1.3.3 or later. On OpenShift with GreyMatter, Tool Shed tools also no longer receive any cookies, an `Authorization` header, or an `X-App-Session-Token` header. A custom tool that read them stops working, and neither `core.auth.runtimeCookies.enforce: false` nor `core.auth.runtimeCookies.trustedRuntimeTargets` restores them. See App Garden apps no longer receive the sign-in cookies (see online documentation).
- **VRAM plugin installed in the `kamiwaza` namespace:** the 1.3.4 VRAM plugin chart refuses to install into `kamiwaza` or `kamiwaza-extensions`. Move the plugin to its own namespace, as described in Move a plugin installed in the kamiwaza namespace (see online documentation).
- **OpenShift with Helm 4 and `global.imagePullSecrets` set:** keep the same pull Secret name during the upgrade, and change only the Secret's contents. Renaming it makes the server-side apply conflict with OpenShift's service account controller on the ServiceAccounts' `.imagePullSecrets`. Helm marks the release failed, and it is left partly applied, although the migrations ran and the workloads rolled. To recover, put the old name back and rerun the upgrade, or run one upgrade with `--force-conflicts`.
- **Helm 4, where Kamiwaza's `core-ray` or `core-scheduler` Role was changed with kubectl** (for example, to let it list events): 1.3.4 changes the rules of both Roles, so a server-side apply upgrade stops with a conflict on `.rules`. Check first, and choose a fix, as described in Upgrading with Helm 4: changes made with kubectl (see online documentation).
- **An execution gate or dataset gate that uses your own class:** Kamiwaza now loads a gate class only from its own gate modules, an installed gate package, or a package listed in `AUTH_GATE_CLASSPATH_PREFIXES`. Before, a class set in the Helm value, in the `AUTHZ_EXECUTION_GATE` variable, or in a dataset's properties was loaded without that check. If an execution gate's class is not allowed, every job submission is refused with `403 gate_load_error`. A dataset whose gate class is not allowed is not served. Check both gates as described in Before you start (see online documentation), and list the package before you upgrade, as described in Allow your gate's classpath (see online documentation).

Action required after upgrading

- **Redeploy each tenant-mode vLLM model that needs tool calling.** A deployment gets tool calling only when it is deployed on 1.3.4 or later.
- **Tenant-mode GPU nodes:** the cluster owner turns on the node setting that refuses GPU requests made through environment variables. Tenant-mode GPU serving requires it. If you share GPUs with the VRAM plugin, first upgrade the plugin with this release's chart: on a node with the setting, the 1.3.3 plugin's shares start without a GPU. See Refuse GPU requests by environment (see online documentation).
- **Kubernetes with Istio and OpenShift with GreyMatter:** replace the ForwardAuth signing secret, unless only administrators you trust could deploy App Garden apps or tools. Before 1.3.4, a user who could deploy one could read it. See Rotate the ForwardAuth signing secret (see online documentation).
- **An extension credential an administrator entered before 1.3.4:** rotate it. App Garden deployments created before 1.3.4 keep their extension credentials as plain environment variables, which anyone who can read pods or ConfigMaps in Kamiwaza's namespace can read. Catalog defaults need no action. See Extension credentials from before 1.3.4 (see online documentation).
- **OmniParse, ontology, and VectorDB instances in workrooms:** the upgrade removes the settings that only administrators can now set, such as endpoints, API keys, TLS settings, proxies, and trust stores, from OmniParse, ontology, and VectorDB instances in workrooms other than the Global Workroom, including values an administrator set. When Core first starts after the upgrade, it also removes them from

those instances' running pods, which restart once. An administrator must check Core's log for instances it did not finish, and set the intended values again; a value set again does not reach a running pod. See Workroom context settings removed by the upgrade (see online documentation).

Security

- On Kubernetes with Istio and on OpenShift with GreyMatter, App Garden apps no longer receive the Kamiwaza sign-in cookies with each request, unless an administrator trusts the app. Before, any app a user opened could call Kamiwaza as that user. On OpenShift with GreyMatter, Tool Shed tools also no longer receive any cookies, an `Authorization` header, or an `X-App-Session-Token` header.
- Where Kamiwaza removes its sign-in cookies from a request, it now also ends a cookie at a comma, so a combined `Cookie` header cannot hide a sign-in cookie inside another cookie's value. A cookie value that contains a comma, which RFC 6265 does not allow, reaches the workload split at the comma: `theme="a,b"` arrives as `theme="a; b"`. This applies on Envoy ingress without Istio to apps and tools, and on Kubernetes with Istio and on OpenShift with GreyMatter to App Garden apps that are not trusted with the session.
- App Garden environment values can no longer use `{forwardauth_signature_secret}`; a deployment that does is refused with an error that says why. Before, a user who could deploy an app could read the platform's ForwardAuth signing secret through the API.
- New App Garden deployments keep an extension's credentials in Kubernetes Secrets instead of the pod specification. App Garden deployment responses no longer return credentials: credential-named environment values are left out, and passwords in URLs and compose text are masked. Template responses do the same for users who are not administrators. Administrators, who author templates, still get the template text as written.
- Disabling or deleting a user who is linked to a Keycloak account now disables that account, ends its sessions, and revokes its API tokens. Before, the user's existing credentials kept working. Users disabled or deleted on an earlier release are refused by Kamiwaza, but their Keycloak accounts stay enabled until you repeat the change; see Users disabled or deleted before the upgrade (see online documentation).
- A user who is linked to a Keycloak account can no longer be renamed; the request returns `409`. Renaming one and then resetting its password created a second Keycloak account that disabling or deleting the user did not reach.
- Users who are not administrators can set only the model selection, enable, timeout, and processing settings of an OmniParse instance. Endpoints, API keys, TLS settings, and every other key need an administrator; anyone else gets `403`.
- A dataset create or update request that sets or changes a gate key (`gate`, `gate_type`, or `gate_*`) in `properties` is refused with `400`. Bind or remove a dataset gate with `PUT` or `DELETE` `/api/catalog/datasets/{urn}/gate`.
- A runtime route (`/runtime/apps/<name>` or `/runtime/tools/<name>`) in a workroom you cannot access returns `404 runtime_target_not_found` instead of `403`. On Envoy ingress without Istio, that is the same answer as for a name that does not exist.
- On Kubernetes with Istio, the access check on model and tool routes runs even if the step that decides whether to check fails, and model engines can reach only the access check on Core's API.
- The cluster owner of a tenant-mode installation can anchor the signed profile on the `kamiwaza` namespace, so the Helm values cannot replace the owner's key or put back an earlier profile, and can cap the accelerators the namespace can request. Both use `owner_tool` from the Core image, are optional in 1.3, and are planned to be required from 1.4. See Anchor the profile on the namespace (see online documentation) and Cap the namespace's accelerators (see online documentation).
- On Kubernetes with Istio, a cluster administrator can apply a pod security guard that enforces the Pod Security `baseline` controls in Kamiwaza's namespaces. On an existing installation, apply it in warn mode first. See Pod security guard (see online documentation).
- Users can create their own VectorDBs in the Global Workroom. Only the owner and administrators can see or use one; anyone else gets `404`. Owners share one through `/api/authz/resources/context_vectordb/grants`.

- The shared Global Milvus, `milvus-global-workroom`, belongs to the system, and Global ingest and search always use it. In 1.3.3, the first user to ingest into the Global Workroom became its owner. Kamiwaza now releases that owner on startup. Creating a Global OmniParse instance still needs an administrator.
- `POST /api/context/ontologies/{id}/graph/bulk` with `"mode": "replace_group"` in the request body deletes the whole target graph first, so only the workroom owner (in the Global Workroom, the ontology instance's owner) or an administrator may send it. Anyone else gets `400` with code `replace_group_requires_owner`, and must use `"mode": "merge"`, which upserts.
- Service and connector extensions, such as `service-graphiti`, accept connections only from Core and from the callers their template declares, on every routing provider. Before, on Kubernetes with Istio and on OpenShift with GreyMatter, any App Garden app could call them directly. Existing deployments are narrowed after the upgrade: each one's main pod restarts once, on its own, to close old connections. A service whose pod is replaced rather than rolled, such as standalone Milvus, is briefly unavailable. Calling a service extension's pods directly from another app is not supported.

Fixes and improvements

- After a platform upgrade, App Garden selections below the release's extension versions, including ones an administrator chose, move up to those versions. New deployments then use the versions the release ships. Running deployments are not changed.
- An extension whose image cannot be pulled fails, and its error names the image and the registry's reason. Before, it waited out the readiness timeout.
- An App Garden app whose image takes longer than 10 minutes to pull is no longer marked failed. A launch that does fail removes the app's Kubernetes workloads.
- Tenant-mode vLLM supports tool calling. A deployment whose chat template accepts tools starts with `--enable-auto-tool-choice` and the tool-call parser for its model family. `tool_call_parser` and `enable_auto_tool_choice` in the model configuration override that.
- After a Ray head restart, an external model deployment whose route is missing waits for Ray and recovers under the same ID, instead of stopping.
- On Envoy ingress without Istio, runtime routes say why they refuse a request: a route that is still deploying returns `409` with `Retry-After`, and an access check that cannot complete returns `503` with a reason, instead of an empty `403`.
- A request to a `/runtime/` path that has no route returns `404` instead of `200`.
- Requests that cannot work say why. A model registration that puts external-endpoint settings outside `external_endpoint`, and `POST /api/apps/deploy_app` without `template_id`, are refused with `422` instead of failing later. A refused App Garden environment value returns `400` with the reason.
- Names that start with `kz-visibility-stand-in-` are reserved. Creating an extension or app deployment with one is refused with `422`.
- The Settings page shows the Kamiwaza version, and the build's commit when you hover over or focus it.
- On OpenShift with `global.imagePullSecrets` set, a Helm 4 server-side apply upgrade no longer fails with a conflict on the ServiceAccounts' `imagePullSecrets`.
- The VRAM plugin chart refuses to install on a cluster whose NVIDIA GPU Operator runs its container toolkit in CDI mode, where shares cannot start. VRAM sharing needs the toolkit installed on the host; see GPU Nodes (see online documentation).
- Kaizen (Legacy) 2.1.8 sends model calls through Kamiwaza's authenticated model routes by default, instead of directly to the model server.
- Kaizen 0.7.11 runs its code sandbox on OpenShift clusters whose security policy assigns Pod user and group IDs, such as the default `restricted-v2`. Before, code execution never started there and members were told the sandbox was busy. If the cluster still refuses to start sandbox workers, members are told code execution is unavailable, and a Kaizen administrator sees the cluster's reason on the Kaizen Setup page.
- The Google and Microsoft 365 connector images are published with the release. In 1.3.2 and 1.3.3 they were missing from the release, so the connectors could not be deployed.

- The air-gapped field kit's release includes every file its README uses: the `INTAKE-<version>.json` intake record, the pickup runbook, and the files bundle. The 1.3.3 kit could not be followed as written.
- `POST /api/context/ontologies/{id}/graph/bulk` writes many graph records in one request, through Core. It needs `service-graphiti` 2.7.0 or later, and returns `501` against an earlier version. Core refuses a body larger than `CONTEXT_SERVICE_GRAPH_BULK_MAX_BODY_BYTES` (default 16 MiB, the same as `service-graphiti`'s own limit) with `413 graph_bulk_body_too_large`, without reading more than the limit. A busy graph writer, an unavailable embedding model, or a slow upload returns `503` with `Retry-After`; retry it.
- Writable datasets can be created in the Global Workroom. Before, they needed a workroom.
- Extension versions: Kaizen 0.7.11, Kaizen (Legacy) 2.1.8, Graphiti 2.7.1, Milvus 2.4.15, Skills Library 0.5.5, OmniParse 2.2.8, and Data Connector Builder and Runtime 0.2.7 (alpha). The other extensions are unchanged from 1.3.3.

1.3.3

Security fixes, Kaizen 0.7.6, and OmniParse 2.2.7.

Before upgrading

- **Kubernetes with Istio:** `global.authz.extAuthz.serviceHost` must be unset, or name `core-api.<namespace>.svc.<cluster domain>` in Kamiwaza's namespace. Any other value makes `helm upgrade` fail before it changes anything. See Upgrade Kamiwaza (see online documentation).

Action required after upgrading

- **Redeploy every tenant-mode model.** Models deployed earlier do not get the new model-route authentication until redeployed, and diffusion deployments from earlier releases are refused through the gateway until redeployed.
- **Select Kaizen 0.7.6 and OmniParse 2.2.7** in the App Garden, then redeploy OmniParse. Kaizen 0.7.6 is a new deployment; see Upgrade Kamiwaza (see online documentation).
- **Air-gapped installations:** keep the earlier Kaizen images in your registry while deployments of earlier Kaizen versions still run.

Security

- On Kubernetes with Istio, model routes (`/runtime/models/...`) now require credentials. Before, they answered unauthenticated requests through the gateway.
- Direct-runtime tool routes (`/runtime/tools/...`), including MCP tools and OmniParse, now run Kamiwaza's access check. Core updates existing tool routes when it starts.
- Extension network policies admit only each extension's primary port, so an extension's data stores are no longer reachable from the whole namespace.
- Kaizen 0.7.6 generates unique service credentials for each deployment and keeps its sandbox worker token in a Secret instead of the pod specification.
- Updated OmniParse 2.2.7 (mcp 2.3.0), the CUDA diffusion engine (torchvision 0.28.0 with CUDA 12.9, without FFmpeg), and Keycloak (`v26.7.4-kz.4`).
- Rebuilt the vLLM 0.28.0 CUDA engine images with security fixes (`v0.28.0-kz.6`, and `v0.28.0-arm64.kz.5` for GB10).

Fixes and improvements

- Kaizen chat works on Envoy ingress without Istio when an administrator deploys Kaizen from the catalog (`core.auth.modelInference.trustedRuntimeTargets`, default `[template:kaizen]`).
- CUDA diffusion runs on NVIDIA GB10 (arm64).

- Model staging no longer runs out of memory on nodes with slow disks.
- Tenant-mode vLLM pods are no longer evicted while compiling a model.
- App Garden deploy errors name the missing required variable.
- `GET /api/kamiwaza/metadata` returns the running Core version and build.
- Kaizen 0.7.6: Project members can search and cite documents shared with the Project, Kaizen says when it cannot do something instead of failing the turn, a chat that finishes in a hidden tab raises a notification, and TLS certificate errors name the cause and the fix.

1.3.2

Extension and security updates.

- Kaizen 0.7.4, Kaizen (Legacy) 2.1.6, OmniParse 2.2.6, Skills Library 0.5.4, Milvus 2.4.12, Graphiti 2.6.9, Kamiwaza DDE 2.5.5, and Data Connector Builder and Runtime 0.2.6.
- The VRAM plugin chart is published with the release. See GPU Nodes (see online documentation).
- Tenant-mode diffusion uses the refreshed engine image.

1.3.1

Fixes, Kaizen 0.7.3, and security updates. 1.3.1 is the first release with a field kit for air-gapped sites: an intake list of every image, scripts to fetch and verify the release, and values templates for a site registry. If your transfer process changes image digests, install with the values the field kit generates in its tag mode (`IMAGE_REF_MODE=tag`, the default), which refer to every image by tag; its intake check verifies the copies in your registry.

Action required after upgrading

- **Air-gapped installations:** set `core.templates.availability.enabled: false`, as in step 5 of Install Without Internet Access (see online documentation). The field kit includes it as `kamiwaza-offline.yaml`.
- **Kaizen on air-gapped installations:** setting **Sandbox outbound scope** (`SANDBOX_EGRESS_MODE`) to `isolated` is recommended.

Fixes and improvements

- Air-gapped installations no longer fail on the App Garden catalog sync.
- The bundled embedding model loads only from the bundle, and reports a clear error if the file is missing or does not match its checksum.
- On OpenShift with GreyMatter, model deployments wait for GreyMatter, and model pods that started too early are restarted.
- Workroom deletes and membership changes wait for internal locks instead of failing with `503`.
- vLLM on NVIDIA GB10 (DGX Spark) in tenant-mode inference.
- OpenAI-compatible model endpoints other than OpenAI, such as vLLM, Ollama, or LiteLLM, can receive the requested reasoning effort. Turn it on with `supports_reasoning_effort` in the model configuration's `external_endpoint`; see OpenAI-compatible chat (see online documentation).
- Kaizen 0.7.3 (includes 0.7.2):
  - Built-in agent tools work without internet access, and the code sandbox includes a Python data and document stack (pandas, NumPy, SciPy, Matplotlib, openpyxl, python-docx, python-pptx, ReportLab, pypdf).
  - Hybrid search and semantic indexing reach the embedding model reliably.
  - Kaizen lists models on Envoy ingress without Istio.

    - **Tasks are off by default** for new deployments (**Durable Tasks**, `TASKS_ENABLED`). Existing deployments keep their setting.
    - A mobile layout, scoped public web access, and federated cluster tools (off by default; `FEDERATION_TOOLS_ENABLED`).
    - **Multipart upload limit.** By default, at most two multipart upload parts use a database connection at the same time, across all API replicas. Additional parts get a retryable `429`, and a large browser upload under sustained load can use up its retries and fail. Raise `MULTIPART_PART_DB_CONNECTION_BUDGET` only after you confirm that your database has spare connections, and use the same value on every API replica.
    - Reliability and clearer error messages.
- Extension versions: Kaizen 0.7.3, OmniParse 2.2.4, Skills Library 0.5.3, Hello Web 1.0.1.

Security

- Updated the Core, frontend, and Keycloak user-setup base images (Node.js 22, Python 3.12), nginx 1.30.5 in the Hello Web sample, OmniParse 2.2.4 dependencies and base image, the Kaizen 0.7.3 base images, the Skills Library 0.5.3 dependencies, and source-map-js 1.2.2 (GHSA-68fv-2mgg-jv7q (https://github.com/advisories/GHSA-68fv-2mgg-jv7q)).

Installing Kamiwaza

Kamiwaza installs with Helm into a single namespace on a Kubernetes cluster that you run. Installation has two stages, usually done by two roles:

1. **Prepare the platform.** A platform administrator makes sure the cluster provides what Kamiwaza depends on: a namespace, an installer identity, a service mesh and gateway, storage, DNS, and a few recorded facts. Kamiwaza does not set up or modify your cluster. Each environment has its own prerequisites page.
2. **Install Kamiwaza.** Using the namespace-scoped installer identity, you add the license, write a values file with the recorded facts, and run one `helm upgrade --install`.

Before you begin

- Review System Requirements (see online documentation): hardware sizing, supported operating systems, GPU support, and supported configurations.
- Get a **license file** from your Kamiwaza representative. See License File (see online documentation).

Choose your environment

Kamiwaza runs on several kinds of Kubernetes environment, each with its own prerequisites, install, and uninstall pages. Find yours in Installation Environments (see online documentation).

Upgrading from an earlier release

There is no in-place upgrade from Kamiwaza 1.2 or earlier. Contact your Kamiwaza representative to plan an assisted migration to a new installation.

Within a release line, upgrade with `helm upgrade --install`, using the new version and the same values file. See Upgrade Kamiwaza (see online documentation) for the supported upgrades and what to do before and after.

After installation

Verify the platform and log in as described at the end of your installation guide, then see the Quickstart (see online documentation).

To remove Kamiwaza, follow the uninstall page for your environment.

System Requirements

Hardware Requirements

CPU

- **Minimum Cores**: 8+ cores
- **Recommended Cores**: 16+ cores for CPU-based inference workloads
- **Architecture**: x64/amd64 (64-bit)

Memory

System RAM

Deployment	Minimum	Recommended	Notes
Standard	16GB	32GB	Baseline install; limited capacity for apps and tools
Production	32GB	64GB+	Production workloads
GPU Workloads	32GB	64GB+	System RAM alongside GPU vRAM

GPU Memory (vRAM)

- **GPU Inference**: 16GB+ vRAM required
- **Recommended**: 32GB+ vRAM for optimal GPU inference performance

GPU (Optional)

Kamiwaza supports multiple GPU and accelerator platforms:

Discrete GPUs:

- NVIDIA GPUs with compute capability 7.0+ (Linux)
- AMD GPUs via ROCm (Linux) — see Software Dependencies (see online documentation)

In a namespaced install, which serves local models through tenant-mode inference (see online documentation), this release serves NVIDIA GPUs only: no ROCm engine image ships. Prepare GPU nodes as described in GPU Nodes (see online documentation).

Unified Memory Systems:

- **NVIDIA DGX Spark** - GB10 Grace Blackwell, 128GB unified memory
- **AMD Ryzen AI Max+ 395** - "Strix Halo" platform, up to 128GB unified memory

See Special Considerations (see online documentation) for detailed unified memory platform specifications.

Storage

The platform administrator supplies persistent storage before installation. Kamiwaza's volumes consume that storage; Kamiwaza does not install a storage driver. See Block storage in the prerequisites for your configuration: Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation).

Storage Performance

- **Required**: SSD (Solid State Drive)
- **Preferred**: NVMe SSD for optimal performance
- **Minimum**: SATA SSD
- **Note**: Model weights can be on a separate HDD but load times will increase significantly

Storage Capacity

Size each backing filesystem for the peak simultaneous use: downloaded and extracted artifacts, runtime images, model caches and staging, logs, and application volumes located on that filesystem. Replicated storage also needs its configured replica capacity and the provider's reserved free-space margin.

As a planning allowance, give each node that runs Kamiwaza **350 GB free on the filesystem that holds the container runtime's images**, usually `/var/lib`. This covers images, caches, and headroom; it is not a guarantee that Kamiwaza's volumes fit. When the storage provider uses the same disk (for example, `local-path` or Longhorn), add Kamiwaza's volume requests, about 624 GiB by default (see Block storage for Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation)), and the provider's replication and reserve. Size storage on separate disks separately.

Capacity Planning

Component	Minimum	Recommended	Notes
Operating System	20GB	50GB	Ubuntu/RHEL base + dependencies
Kamiwaza	50GB	50GB	Python environment, Ray, services
Model Storage	50GB	500GB+	Depends on number and size of models
Database	10GB	50GB	PostgreSQL for metadata
Vector Database	10GB	100GB+	For embeddings (if enabled)
Logs & Metrics	10GB	50GB	Rotated logs, Ray dashboard data
Scratch Space	20GB	100GB	Temporary files, downloads, builds
Total	Workload-dependent	Workload-dependent	Apply the peak-use calculation above; do not double-count shared caches

The rows describe running workloads. Also allow for simultaneous image import, model staging, and storage-provider replication and reservation overhead.

Storage Performance Requirements

Local Storage (Single Node):

- **Minimum:** SATA SSD (500 MB/s sequential read)
- **Recommended:** NVMe SSD (2000+ MB/s sequential read)
- **Note:** HDD is only recommended for non-dynamic model loads and low KV cache usage - model load times can be very long (15+ minutes); models are in memory after load

Performance Targets:

- **Sequential Read:** 2000+ MB/s (model loading)
- **Sequential Write:** 1000+ MB/s (model downloads, checkpoints)
- **4K Random Read IOPS:** 50,000+ (database, concurrent access)
- **4K Random Write IOPS:** 20,000+ (database writes, logs)

Why It Matters:

- 7B model (14GB): Loads in ~7 seconds on NVMe vs ~28 seconds on SATA SSD
- Concurrent model loads across Ray workers stress random read performance
- Database query performance directly tied to IOPS

Supported Operating Systems

Kamiwaza runs on Kubernetes nodes with these operating systems:

Operating system	Architecture
Red Hat Enterprise Linux 9	x64/amd64
Ubuntu 24.04 LTS	x64/amd64
Ubuntu 22.04 LTS	x64/amd64

On OpenShift, the node operating system is the one OpenShift provides, and support follows the OpenShift version.

Supported Configurations

Configuration	Description	Support
Single-node Kubernetes with Istio	One Kubernetes node with an Istio service mesh and ingress gateway	Preferred
OpenShift with GreyMatter	Customer-operated OpenShift, including managed OpenShift such as ROSA, with GreyMatter routing	Preferred
Air-gapped	No outbound access; images and charts come from your own registry	Preferred
Federation	Two or more Kamiwaza installations joined over the mesh	Preferred
NVIDIA GPU	GPU inference through the NVIDIA GPU Operator or device plugin	Preferred
Multi-node cluster	One Kubernetes cluster spread over several nodes; see Multi-node clusters in the prerequisites for your environment (see online documentation)	Supported

Air-gapped and NVIDIA GPU apply on top of a platform configuration; they are not platforms themselves.

Preferred configurations are the ones Kamiwaza recommends. **Supported** configurations are fully supported and documented.

Operation	Support
Uninstall	Supported. See the uninstall page for your environment (see online documentation).

Software Dependencies

What You Provide

Component	Requirement	Notes
License file	A signed `license.lic` issued by Kamiwaza	See License File (see online documentation). Contact your Kamiwaza representative.
Kubernetes cluster	Prepared to meet Kamiwaza's platform prerequisites	See the prerequisites for your environment (see online documentation)
Supported OS	RHEL 9, Ubuntu 24.04, or Ubuntu 22.04 on every node	See Supported Operating Systems (see online documentation)
Browser	Chrome 141+ (tested and recommended)	Download Chrome (https://www.google.com/chrome/)
GPU drivers	For GPU inference only; see below	Install on GPU nodes before installing Kamiwaza

GPU Drivers (Required for GPU Inference)

Install the appropriate driver for your GPU hardware:

NVIDIA GPUs:

Component	Requirement	Installation guide
NVIDIA driver for CUDA 13 engine images (llama.cpp, vLLM) and DGX Spark	580.65.06 (R580) or later	GPU Nodes (see online documentation)
NVIDIA driver for CUDA 12 engine images (llama.cpp, diffusion)	525.60.13 or later; R570 or later for Blackwell GPUs and for diffusion	GPU Nodes (see online documentation)
NVIDIA Container Toolkit	Required for GPU containers	GPU Nodes (see online documentation)

The engines, their CUDA majors, and the GPUs each one serves are listed in Supported GPUs and driver floors (see online documentation).

DGX Spark software releases include the matching R580 driver and CUDA 13 stack. See the DGX Spark release notes (https://docs.nvidia.com/dgx/dgx-spark/release-notes.html) for the versions in each release.

AMD GPUs (ROCm):

Component	Requirement	Installation guide
ROCm for Ryzen AI Max+ 395 (gfx1151)	ROCm 7.2.1 or later on Ubuntu 24.04	Ryzen native Linux compatibility (https://rocm.docs.amd.com/projects/radeon-ryzen/en/docs-7.2.1/docs/compatibility/compatibilityryz/native_linux/native_linux_compatibility.html)
Other AMD GPUs	A ROCm release that lists the GPU and operating system as supported	ROCm compatibility matrix (https://rocm.docs.amd.com/en/latest/compatibility/compatibility-matrix.html)
Container GPU access	`/dev/kfd` and `/dev/dri` exposed to the container runtime	ROCm containers guide (https://rocm.docs.amd.com/en/latest/how-to/docker.html)

AMD lists the Radeon 8060S in Ryzen AI Max+ 395 systems as production-supported on the ROCm 7.2.x native Linux path. Follow the Ryzen native Linux installation guide (https://rocm.docs.amd.com/projects/radeon-ryzen/en/docs-7.2/docs/install/installryz/native_linux/install-ryzen.html); the retired ROCm 7.10 preview path is not required.

Verifying System Requirements

Use these commands on each node to check GPU access and host resources before installation. They do not replace the checks in the prerequisites for your environment (see online documentation).

NVIDIA GPU (if applicable)

```
# Check NVIDIA driver
nvidia-smi
# Expected for CUDA 13 engine images and DGX Spark: driver 580.65.06 or later,
#   and "CUDA Version: 13.0" or later in the header
# Expected for CUDA 12 engine images: driver 525.60.13 or later, or 570 or later
#   for Blackwell GPUs and for diffusion
# Should display GPU name, driver version, and CUDA version

# Check NVIDIA Container Toolkit
nvidia-ctk --version
# Expected: Any version indicates toolkit is installed
# Example output: NVIDIA Container Toolkit CLI version 1.17.3
```

AMD ROCm (if applicable)

```
# Check ROCm installation
rocm-smi
# Expected: Should display AMD GPU information
# Look for: GPU temperature, utilization, memory usage

# Check ROCm version
cat /opt/rocm/.info/version
# Expected for Ryzen AI Max+ 395 (gfx1151): 7.2.1 or later

# Verify GPU device access
ls -la /dev/kfd /dev/dri
# Expected: Both devices should exist and be accessible
```

System Resources

```
# Check available memory
free -h
# Expected: At least 16GB total (32GB+ recommended). free -h shows a little less
# than installed memory, because the kernel and firmware reserve some: about 15Gi
# on a 16 GB machine and about 30Gi on 32 GB.
# Look for "Mem:" row, "total" column

# Check CPU cores
nproc
# Expected: 8 or more cores

# Check free space on the filesystem that holds container images
df -h /var/lib
# Plan for 350GB free; see Storage Capacity above
```

Hardware Recommendation Tiers

Kamiwaza is a distributed AI platform built on Ray that supports both CPU-only and GPU-accelerated inference. Hardware requirements vary significantly based on:

- **Model size**: From 0.6B to 70B+ parameters
- **Deployment scale**: Single-node development vs multi-node production
- **Inference engine**: LlamaCpp (CPU/GPU), VLLM (GPU)
- **Workload type**: Interactive chat, batch processing, RAG pipelines

GPU Memory Requirements by Model Size

The table below provides real-world GPU memory requirement estimates for representative models at different scales. These estimates assume FP8 and include overhead for context windows and batch processing.

Model Example	Parameters	Minimum vRAM	Notes
GPT-OSS 20B	20B	24GB	Includes weights + 1-batch max context; fits 1x 24GB GPU (e.g., L4/RTX 4090)
GPT-OSS 120B	120B	80GB	~40GB weights + 1-batch max context; 1x H100/H200 or 2x A100 80GB recommended
Qwen 3 235B A22B	235B	150GB	~120GB weights + 1-batch max context; 2x H200 (282GB) or 2x B200 (384GB) ideal for max context
Qwen 3-VL 235B A22B	235B	150GB	Same base minimum (includes 1-batch max context); budget +20-30% vRAM for high-res vision inputs

Key Considerations:

- **Minimum vRAM**: FP8 weights + 1-batch allocation at your target max context
- **Headroom**: For longer contexts, larger batch sizes, and concurrency, budget additional vRAM beyond minimums
- **Vision Workloads**: Image/video processing adds overhead; budget 20-30% more for vision-language models
- **Tensor Parallelism**: Distributing large models (120B+) across multiple GPUs requires high-bandwidth interconnects (NVLink 3.0+)

Tier 1: Development & Small Models

Use Case: Local development, testing, small to medium model deployment (up to 13B parameters)

Hardware Specifications:

- **CPU:** 8-16 cores / 16-32 threads
- **RAM:** 32GB (16GB minimum for development only)
- **Storage:** 400GB NVMe SSD as an initial development allocation, assuming Kamiwaza's volumes are on separate storage. If the volumes share this disk, size it using Storage Capacity (see online documentation).
- **GPU:** Optional - Single GPU with 16-24GB VRAM
  - NVIDIA RTX 4090 (24GB)
  - NVIDIA RTX 4080 (16GB)
  - NVIDIA T4 (16GB)
- **Network:** 1-10 Gbps

Workload Capacity:

- Low-volume workloads: 1-10 concurrent requests (supports dozens of interactive users)
- Development, testing, and proof-of-concept deployments
- Light production workloads

Tier 2: Production - Medium to Large Models

Use Case: Production deployment of medium to large models (13B-70B parameters), high throughput

Hardware Specifications:

- **CPU:** 32 cores / 64 threads
- **RAM:** 128-256GB system RAM
- **Storage:** 1.2–2TB NVMe SSD as a planning range; size the model library, local PVCs, and replication reserves using Capacity Planning (see online documentation).
- **GPU:** 1-4 GPUs with 40GB+ VRAM each
  - 1-4x NVIDIA B200 (192GB HBM3e)
  - 1-4x NVIDIA H200 (141GB HBM3e)
  - 1-4x NVIDIA RTX 6000 Pro Blackwell (48GB)
  - 1-2x NVIDIA H100 (80GB)
  - 1-4x NVIDIA A100 (40GB or 80GB)
  - 1-2x NVIDIA L40S (48GB)
  - 2-4x NVIDIA A10G (24GB) for tensor parallelism
- **Network:** 25-40 Gbps

Workload Capacity:

- Medium-scale production: 100s to 1,000+ concurrent requests (supports thousands of interactive users)
- Example: Per-GPU batch size of 32 across 8 GPUs = 256 concurrent requests; batch size of 128 = 1,024 requests
- Production chat applications
- Complex RAG pipelines with embedding generation
- Batch inference

Tier 3: Enterprise Multi-Node Cluster

Use Case: Enterprise deployment with multiple models, high availability, horizontal scaling, 99.9%+ SLA

Cluster Architecture:

Head Node (Control Plane):

- **CPU:** 16 cores / 32 threads
- **RAM:** 64GB
- **Storage:** 500GB NVMe SSD as an initial control-plane allocation; verify local PVC and staging requirements separately.
- **GPU:** Same class as worker nodes (homogeneous cluster recommended)
- **Role:** Ray head, API gateway, scheduling, monitoring (head performs minimal extra work; Ray backend load is distributed across nodes)

Worker Nodes (3+ nodes for HA):

- **CPU:** 32-64 cores / 64-128 threads per node
- **RAM:** 256-512GB per node

- **Storage:** 2TB NVMe SSD per node (local cache)
- **GPU:** 4-8 GPUs per node (same class as head node)
- **Network:** 40-100 Gbps (InfiniBand for HPC workloads)

> Note: For production clusters, avoid non-homogeneous hardware (e.g., GPU-less head nodes). Each node participates in data plane duties (ingress gateway, HTTP proxying, etc.), so matching GPU capabilities simplifies scheduling and maximizes throughput.

Shared Storage:

- High-performance NAS or distributed filesystem (Lustre, CephFS)
- 10TB+ capacity, NVMe-backed
- 10+ GB/s aggregate sequential throughput
- Low-latency access (< 5ms) from all nodes

Workload Capacity:

- Multiple models deployed simultaneously
- High-scale production: 1,000–10,000+ concurrent requests (supports tens of thousands of interactive users)
- Batch sizes scale with GPU count and model size; smaller requests enable higher throughput per GPU
- High availability with automatic failover
- Horizontal auto-scaling based on load
- Production SLAs (99.9% uptime)

Cloud Provider Instance Mapping

AWS EC2 Instance Types

Tier	Instance Type	vCPU	RAM	GPU	Storage
Tier 1: CPU-only	`m6i.2xlarge`	8	32GB	None	400GB gp3
Tier 1: With GPU	`g5.xlarge`	4	16GB	1x A10G (24GB)	400GB gp3
Tier 1: Alternative	`g5.2xlarge`	8	32GB	1x A10G (24GB)	400GB gp3
Tier 2: Multi-GPU	`g5.12xlarge`	48	192GB	4x A10G (96GB)	2TB gp3
Tier 2: Alternative	`p4d.24xlarge`	96	1152GB	8x A100 (320GB)	2TB gp3
Tier 3: All Nodes	`p4d.24xlarge`	96	1152GB	8x A100 (320GB)	2TB gp3

Notes:

- Tier 1 disk figures are planning allocations and assume Kamiwaza's volumes are on separate storage. If the volumes share this disk, size it using Storage Capacity (see online documentation).
- Use `gp3` SSD volumes (not `gp2`) for better performance/cost
- For Tier 3 shared storage: Amazon FSx for Lustre or EFS (with Provisioned Throughput)
- Use Placement Groups for low-latency multi-node clusters (Tier 3)
- H100 instances (`p5.48xlarge`) available in limited regions for highest performance
- Latest options: Emerging `p6` / `p6e` families with H200/B200/Grace-Blackwell are rolling out in select regions; map to Tier 2/3 as available.

Google Cloud Platform (GCP) Instance Types

Tier	Machine Type	vCPU	RAM	GPU	Storage
Tier 1: CPU-only	`n2-standard-8`	8	32GB	None	400GB SSD
Tier 1: With GPU	`n1-standard-8` + `1x T4`	8	30GB	1x T4 (16GB)	400GB SSD
Tier 1: Alternative	`g2-standard-8` + `1x L4`	8	32GB	1x L4 (24GB)	400GB SSD
Tier 2: Multi-GPU	`a2-highgpu-4g`	48	340GB	4x A100 (160GB)	2TB SSD
Tier 2: Alternative	`g2-standard-48` + `4x L4`	48	192GB	4x L4 (96GB)	2TB SSD
Tier 3: All Nodes	`a2-highgpu-8g`	96	680GB	8x A100 (320GB)	2TB SSD

Notes:

- Tier 1 disk figures are planning allocations and assume Kamiwaza's volumes are on separate storage. If the volumes share this disk, size it using Storage Capacity (see online documentation).
- Use `pd-ssd` or `pd-balanced` persistent disks (not `pd-standard`)
- For Tier 3 shared storage: Filestore High Scale tier (up to 10 GB/s)
- Use Compact Placement for low-latency multi-node clusters (Tier 3)
- L4 GPUs (24GB) available as cost-effective alternative to A100
- Latest options: Blackwell/H200 classes are entering preview/limited availability; consider AI Hypercomputer offerings as they launch.

Microsoft Azure Instance Types

Tier	VM Size	vCPU	RAM	GPU	Storage
Tier 1: CPU-only	`Standard_D8s_v5`	8	32GB	None	400GB Premium SSD
Tier 1: With GPU	`Standard_NC4as_T4_v3`	4	28GB	1x T4 (16GB)	400GB Premium SSD
Tier 1: Alternative	`Standard_NC6s_v3`	6	112GB	1x V100 (16GB)	400GB Premium SSD
Tier 2: H100 (recommended)	`Standard_NC40ads_H100_v5`	40	320GB	1x H100 (80GB)	2TB Premium SSD
Tier 2: H100 Multi-GPU	`Standard_NC80adis_H100_v5`	80	640GB	2x H100 (160GB)	2TB Premium SSD
Tier 2: A100 Multi-GPU	`Standard_NC96ads_A100_v4`	96	880GB	4x A100 (320GB)	2TB Premium SSD
Tier 2: A100 Alternative	`Standard_NC48ads_A100_v4`	48	440GB	2x A100 (160GB)	2TB Premium SSD
Tier 3: H100 (recommended)	`Standard_ND96isr_H100_v5`	96	1900GB	8x H100 (640GB)	2TB Premium SSD
Tier 3: A100 Alternative	`Standard_ND96asr_v4`	96	900GB	8x A100 (320GB)	2TB Premium SSD

Notes:

- Tier 1 disk figures are planning allocations and assume Kamiwaza's volumes are on separate storage. If the volumes share this disk, size it using Storage Capacity (see online documentation).
- Use Premium SSD (not Standard HDD or Standard SSD)
- For Tier 3 shared storage: Azure NetApp Files Premium or Ultra tier
- Use Proximity Placement Groups for low-latency multi-node clusters (Tier 3)
- NDm A100 v4 series offers InfiniBand networking for HPC workloads
- Latest options: Blackwell/H200-based VM families are announced/rolling out; align Tier 2/3 to those SKUs where available.

Network Configuration

Network Bandwidth Requirements

Single Node Deployment

Network Bandwidth:

- **Minimum:** 1 Gbps (for model downloads, API traffic)
- **Recommended:** 10 Gbps (for high-throughput inference)

Considerations:

- Internet bandwidth for downloading models from HuggingFace (one-time)
- Client API traffic for inference requests/responses
- Monitoring and logging egress

Multi-Node Cluster

Inter-Node Network:

- **Minimum:** 10 Gbps Ethernet
- **Recommended:** 25-40 Gbps Ethernet or InfiniBand
- **Latency:** < 1ms between nodes (same datacenter/availability zone)

Why It Matters:

- Ray distributed scheduling requires low-latency communication
- Tensor parallelism transfers large model shards between GPUs
- Shared storage access impacts model loading performance

Network Ports

Inbound: users reach Kamiwaza only through the platform's ingress gateway, on 443/tcp (HTTPS). Port 80/tcp is optional, for redirecting to HTTPS. Models and apps are served through the same gateway; they need no ports of their own.

Outbound: nodes and pods need HTTPS (443/tcp) access to:

- `ghcr.io` and `pkg-containers.githubusercontent.com`, for Kamiwaza charts and images;
- the model sources you use, for example `huggingface.co`, or your cloud model provider's endpoint.

Air-gapped installs have no outbound requirement.

Special Considerations

NVIDIA DGX Spark

The NVIDIA DGX Spark is a compact AI workstation powered by the GB10 Grace Blackwell Superchip:

- **CPU:** 20-core ARM (10x Cortex-X925 + 10x Cortex-A725)
- **GPU:** Blackwell architecture with 6,144 CUDA cores
- **Memory:** 128GB LPDDR5x unified memory (273 GB/s bandwidth)
- **AI Compute:** Up to 1 PFLOP FP4 AI performance
- **Storage:** 4TB NVMe SSD
- **Networking:** Dual QSFP ports (up to 200 Gbps aggregate)

Capabilities:

- Run models up to 200B parameters locally
- Two interconnected units can handle models up to 405B parameters
- Unified memory architecture eliminates GPU vRAM constraints

AMD Ryzen AI Max+ 395 "Strix Halo"

AMD's Strix Halo platform provides powerful AI inference in a compact form factor:

- **CPU:** 16-core Zen 5 (up to 5.1 GHz), 80MB cache
- **GPU:** Radeon 8060S iGPU (40 CUs, RDNA 3.5 architecture)
- **NPU:** 50 TOPS XDNA 2 neural engine
- **Memory:** Up to 128GB LPDDR5x unified memory (up to 112GB GPU-allocatable)
- **AI Performance:** 126 TOPS total
- **TDP:** 55W (highly power efficient)

Capabilities:

- Run 70B+ parameter models locally
- Available in mini PCs and high-end laptops
- Unified memory architecture (shared CPU/GPU memory)

Shared Storage (Multi-Node Clusters)

Network Filesystem Requirements:

- **Protocol:** NFSv4, Lustre, CephFS, or S3-compatible object storage
- **Network Bandwidth:** 10 Gbps minimum, 40+ Gbps for production
- **Network Latency:** < 5ms between nodes and storage
- **Sequential Throughput:** 5+ GB/s aggregate (10+ GB/s for large clusters)

Object Storage (Alternative):

- S3-compatible API (AWS S3, GCS, MinIO, etc.)
- Local caching layer recommended for frequently accessed models
- Consider bandwidth costs for cloud object storage

Shared Storage Options:

Solution	Use Case	Throughput	Cost Profile
NFS over NVMe	Small clusters (< 5 nodes)	1-5 GB/s	Low (commodity hardware)
AWS FSx for Lustre	AWS multi-node clusters	1-10 GB/s	Medium (pay per GB/month + throughput)
GCP Filestore High Scale	GCP multi-node clusters	Up to 10 GB/s	Medium-High
Azure NetApp Files Ultra	Azure multi-node clusters	Up to 10 GB/s	High
CephFS	On-premises clusters	5-20 GB/s	Medium (requires Ceph cluster)
Object Storage + Cache	Cost-optimized	Varies	Low storage, high egress

Version Compatibility

- NVIDIA driver: 525.60.13 or later for CUDA 12 images (R570 or later for Blackwell GPUs and for diffusion); 580.65.06 (R580) or later for CUDA 13 images and DGX Spark

Important Notes

- **GPU Support**: NVIDIA GPUs (CUDA) and AMD GPUs (ROCm) on Linux nodes
- **Platform**: Kamiwaza does not install or configure Kubernetes, the service mesh, storage drivers, or GPU drivers; see the prerequisites for your environment (see online documentation)
- **Network**: Users reach Kamiwaza through the ingress gateway on 443/tcp; see Network Ports (see online documentation)
- **License**: A license file is required for every installation; see License File (see online documentation)

License File Installation

Every Kamiwaza installation runs with a signed **license file** issued by Kamiwaza. This page covers installing the file on a cluster, confirming the platform sees it, rotating it, and what the startup messages mean. A license file is required for every installation.

What the license file is

- A single file, `license.lic`, issued to your organization by Kamiwaza.
- Signed. The platform verifies the signature at startup with a key built into the Kamiwaza images. Nothing is sent to Kamiwaza and no network access is needed to validate it, so it works in air-gapped environments.
- Carries who the license was issued to, the edition, and an optional expiry date for the commercial term.

Do not edit the file. Any change to its contents, including a stray newline added during copy-paste, invalidates the signature.

If you do not have a license file, contact your Kamiwaza representative or get in touch at https://www.kamiwaza.ai/contact (https://www.kamiwaza.ai/contact).

Install the license file

1. Create a Secret from the file. The Secret must contain exactly one key named `license.lic`:

```
kubectl create secret generic kamiwaza-license \
  --namespace kamiwaza \
  --from-file=license.lic=./license.lic
```

2. Point the chart at the Secret in your Helm values file. The license setting belongs under `core:`:

```
core:
  license:
    existingSecret: kamiwaza-license
```

Then install or upgrade as usual: see Installing Kamiwaza (see online documentation). Kamiwaza mounts the Secret as a whole directory at `/app/licenses` (the `core.license.mountPath` default) and reads `/app/licenses/license.lic`. Keep the directory mount: a `subPath` mount would never receive a rotated license.

Separately from the license file, the chart requires the EULA to be accepted before it renders anything: set `global.eula.accepted: true` in your values. If an install stops with an EULA message, this is the value it is asking for.

Enforcement

Every published Kamiwaza core image is a release build and always requires a valid license. Enforcement is compiled into the release artifact rather than configured by the chart, so there is no chart setting that weakens this policy. Kamiwaza core refuses to start on any license problem listed under Troubleshooting (see online documentation). A passed commercial term is not one of those startup failures.

Verify the license is active

Every API response from the platform carries two headers:

Header	Values
`x-kamiwaza-license-state`	`valid`, `expiring` (term ends within 30 days), or `expired`
`x-kamiwaza-license-expires`	The term end date as `YYYY-MM-DD`; absent when the license has no expiry

```
curl -skI https://<your-kamiwaza-host>/api/ | grep -i x-kamiwaza-license
```

`-k` accepts an edge certificate your workstation does not trust, such as a self-signed one.

The web UI shows the same information: an amber banner when the term ends within 30 days (dismissible), and a red banner once it has ended. **An expired commercial term never stops the platform.** The banner is the only effect; contact your Kamiwaza representative to renew.

If no license headers are present, verify that you are querying a path that goes through the Kamiwaza API. A release build cannot start without a valid license.

Rotate or renew a license

1. Replace the Secret's contents with the new file:

   ```
   kubectl create secret generic kamiwaza-license \
     --namespace kamiwaza \
     --from-file=license.lic=./license.lic \
     --dry-run=client -o yaml | kubectl apply -f -
   ```

2. In your values file, change `core.license.rolloutKey` to any new value (a date works) so the scheduler pod restarts and re-reads the file, and set `core.rayServe.forceRedeployOnStartup: true` so the model-serving layer is redeployed on that restart:

```yaml
core:
  license:
    existingSecret: kamiwaza-license
    rolloutKey: "2026-09-01"
  rayServe:
    forceRedeployOnStartup: true
```

Without `forceRedeployOnStartup`, the scheduler leaves a healthy serving deployment alone, and the serving layer keeps reporting the **old** license state even though the new file was accepted. A renewal that "did not take" in the banner is almost always this.

3. Rerun `helm upgrade --install` with the updated values file. Afterward, remove `forceRedeployOnStartup` from your values file; its default is `false`.
4. Re-check the `x-kamiwaza-license-*` headers.

Troubleshooting

When the license check fails, the `core-scheduler` pod log contains one block that starts with `License check failed (<condition>)`, names the file path it looked in, and ends with a `Condition:` line you can search for. On release builds, core exits after logging this block, and the API stops serving with it: on refusal the platform also retires the Ray Serve application, so you lose the API as well as the scheduler.

Condition	Meaning	What to do
`license_file_missing`	No file at `/app/licenses/license.lic`	Create the Secret with key `license.lic` and set `core.license.existingSecret`, or request a license
`license_file_unreadable`	The file exists but could not be read	Check the Secret holds the complete file and is mounted as a directory, not a `subPath`
`license_tampered`	The signature does not verify	The file was modified or truncated in transit. Request a fresh copy; do not edit license files
`license_wrong_account`	Issued by a different vendor account	This file is not a Kamiwaza-issued license
`license_wrong_product`	Issued for a different Kamiwaza product	Check which product the file was issued for and request the right one
`license_suspended`	The license has been suspended by Kamiwaza	Contact Kamiwaza licensing support
`license_file_stale`	The signed file itself carries an expiry that has passed (distinct from the commercial term, which never blocks startup)	Request a re-issued file
`license_claims_invalid`	A field in the file is malformed, or the file uses a newer format than this Kamiwaza version understands	If the message says the file's schema version is above the maximum this build supports, upgrade Kamiwaza or request a file issued for your version; for any other detail, request a re-issued file — upgrading will not help
`license_gate_misconfigured`	The image itself is missing its built-in trust data	Redeploy a correctly published Kamiwaza image; changing the license file cannot fix this

There is deliberately no `license_expired` condition: an ended commercial term is reported through the headers and banner above, never through the startup check.

Getting a license

Contact your Kamiwaza representative, or get in touch at https://www.kamiwaza.ai/contact (https://www.kamiwaza.ai/contact). Deliver the returned `license.lic` to the cluster as described above; it does not need to be placed anywhere else.

Install Without Internet Access

An air-gapped install is an ordinary install that pulls from your own registry. This page covers the two things that differ: copying the release into a registry your cluster can reach, and the values that point Kamiwaza at it.

Do this page before you install. The checks in the prerequisites for your environment (see online documentation) use two public images that step 3 copies, so on a cluster that already has no internet access, do steps 2 and 3 first (and step 4, if your registry needs credentials), then run those checks, then continue. How you move files across the gap is up to you and your site's transfer process; the commands below assume one machine that can reach both the release registry and yours, and they work the same through any intermediate hop that preserves image digests.

Before you start

You need	Notes
A container registry your cluster's nodes can pull from	Any OCI registry. It must accept repository paths several segments deep, for example `kamiwaza-ai/releases/kamiwaza/images/core`. On a registry that groups repositories into projects, such as Harbor, create a project named `kamiwaza-ai`.
About 20 GB free in that registry	For the images without GPU inference engines. The GPU engine images are several GB each; add them if you will serve models on GPUs.
A tool that copies images between registries	It must copy every platform of a multi-platform image and keep each image's digest unchanged. The examples use `skopeo` with `--all --preserve-digests`; `crane copy` and `oras cp -r` also work.
`yq` 4 (mikefarah/yq (https://github.com/mikefarah/yq)), on the machine you copy from	Turns the release manifest into the image list in step 1. The Python tool also named `yq` takes different arguments.
Helm 3.14 or later, on the machine you copy from	Checks the copied chart and renders it against your registry before you install.
A cluster whose own images pull without internet access	Kamiwaza's release covers only Kamiwaza. The cluster's own components, including your service mesh's sidecar image, must already pull from inside your network. On OpenShift with GreyMatter, you also copy the GreyMatter proxy image that Kamiwaza's Ray pods run; see step 3.

The examples use these names. Set them once in the shell you copy from:

```
SOURCE=ghcr.io/kamiwaza-ai/releases/kamiwaza        # where Kamiwaza publishes the
release
REGISTRY=<your registry host>                       # bare host[:port], no scheme or
path
```

Step 1: Get the image list

The release is one Helm chart plus the images it deploys, and the chart of the optional VRAM plugin (see online documentation) that a cluster administrator installs beside it. Beside the charts, the release publishes a manifest that names every image the release can deploy, at its location in the release and with the tags it is published under: the platform, its databases, the default extensions, the model serving engines, and the default embedding model.

Fetch the manifest and turn it into a list with one image per line:

```
skopeo copy docker://$SOURCE/manifest:1.3.4 dir:./release-manifest
layer=$(yq -r '.layers[0].digest | sub("^sha256:"; "")' ./release-
manifest/manifest.json)
yq '(.components[].artifacts[] | select((.tags // []) | length == 0)
       | .repository + "@" + .digest),
    (.components[].artifacts[] | select((.tags // []) | length > 0)
       | .repository + ":" + .tags[] + "@" + .digest)' \
  "./release-manifest/$layer" > images.txt
```

The manifest is an OCI artifact with one YAML file in it, which `skopeo` saves under its digest. With `oras`, `oras pull $SOURCE/manifest:1.3.4` saves the same file as `release-1.3.4.yaml`.

Each line of `images.txt` is one image at the release location, pinned by digest. Most lines carry the tag the image is published under, for example `<release location>/images/core:<tag>@sha256:<digest>`, and an image published under two tags has a line for each. From 1.3.1, every line carries a tag: an image that Kamiwaza pulls only by digest, such as a model serving engine, is published under `release-<version>-` followed by the first 12 hexadecimal characters of its digest. Lists from earlier releases name such images without a tag: `<release location>/images/<name>@sha256:<digest>`. The copy loop in step 3 handles both.

To leave out the GPU inference engines on a cluster without GPUs, filter the list now:

```
grep -vE 'vllm|cuda|rocm|diffusion' images.txt > images.cpu.txt && mv images.cpu.txt
images.txt
```

Check:

```
wc -l < images.txt
# Expect a count above 0
grep -vE "^$SOURCE/[a-z0-9._/-]+(:[A-Za-z0-9_][A-Za-z0-9._-]*)?@sha256:[0-9a-f]{64}\$"
images.txt
# Expect no output: every line is an image at the release location, by digest
```

Step 2: Trust your registry's certificate

If your registry uses a certificate from a public certificate authority, skip this step.

If it uses a certificate from your own certificate authority, everything that talks to the registry must trust that authority, or it fails with `x509: certificate signed by unknown authority`:

- the machine you copy from, before step 3;
- the machine you run `helm` from, if it is a different one;
- **every node** in the cluster, for its container runtime. The platform administrator does this.

On most distributions, adding the authority to the operating system's trust store is enough:

```
# RHEL 9
sudo cp my-ca.crt /etc/pki/ca-trust/source/anchors/
sudo update-ca-trust

# Ubuntu
sudo cp my-ca.crt /usr/local/share/ca-certificates/my-ca.crt
sudo update-ca-certificates
```

On each node, then restart the container runtime so it reloads the trust store. Where the runtime is a separate service, restart it, for example `sudo systemctl restart containerd`. Where your Kubernetes distribution runs the runtime itself, restart that service instead, for example `sudo systemctl restart k0scontroller` (or `k0sworker`) on k0s. On a single control-plane node this briefly interrupts the Kubernetes API.

On OpenShift, do not change the nodes' trust store. Give the authority to the cluster's image configuration instead, as the platform administrator. Create a ConfigMap in `openshift-config` with one key per registry, named after the registry's host (write `host:port` as `host..port`), holding the authority's certificate. The image configuration names a single ConfigMap, so check first whether it already names one:

```
oc get image.config.openshift.io/cluster -o jsonpath='{.spec.additionalTrustedCA.name}
{"\n"}'
```

If that prints a name, add your registry's key to that ConfigMap, so the registries it already trusts stay trusted:

```
# For registry.example.com:5000; for a registry without a port, the key is the
# host alone, for example registry.example.com
oc -n openshift-config set data configmap/<name it printed> \
  --from-file=registry.example.com..5000=my-ca.crt
```

If it prints nothing, create the ConfigMap and point the image configuration at it:

```
oc -n openshift-config create configmap registry-ca \
  --from-file=registry.example.com..5000=my-ca.crt
oc patch image.config.openshift.io/cluster --type merge \
  -p '{"spec":{"additionalTrustedCA":{"name":"registry-ca"}}}'
```

The nodes pick it up within a few minutes, without restarting. On a managed OpenShift whose provider owns the image configuration, set the same authority through the provider's tooling instead.

Check on each of those machines:

```
curl -sS -o /dev/null -w '%{http_code}\n' https://<your registry host>/v2/
# Expect 200, or 401 if the registry requires credentials. A certificate error
# means this machine does not trust the authority yet.
```

On OpenShift, run this check only on the machines you copy from and run `helm` from. The image configuration gives the authority to the nodes' container runtime, not to their operating system, so `curl` on a node is not a valid check. The pull check in step 3 is the check for the nodes.

Step 3: Copy the release into your registry

If your registry requires credentials, log in to it with your copy tool, and also with `helm registry login`: Helm keeps its own logins, and it pulls the chart from your registry at install. The release registry needs no login.

Copy each image to the same path in your registry, with only the registry host changed: `$REGISTRY/kamiwaza-ai/releases/kamiwaza/...`. Both the chart and Kamiwaza look for images there. The chart pulls the platform images from it, and when Kamiwaza deploys an App Garden extension it rewrites each of the extension's images to the same location, with the step 5 values. Then copy the charts to their original paths: the product chart, and the VRAM plugin's chart if a cluster administrator will install the plugin. A chart is an OCI artifact and copies the same way as an image.

Copy by digest, to the tags the list names. The chart pulls some images by digest and some by tag, and App Garden extensions pull by tag, so a copy that changes a digest or drops a tag fails to pull. `skopeo` does not accept a tag and a digest together, so the loop below copies each image from its digest to its tag. An image listed without a tag gets one derived from its digest, because a registry needs a tag to accept an image; the digest is unchanged.

```
: > copy-failures.txt
while read -r ref || [ -n "$ref" ]; do
  repo=${ref%@*} digest=${ref##*@}
  case ${repo##*/} in
    *:*) tag=${repo##*:} repo=${repo%:*} ;;
    *) tag=sha256-${digest#sha256:} ;;
  esac
  skopeo copy --all --preserve-digests --retry-times 3 \
    "docker://$repo@$digest" \
    "docker://$REGISTRY/kamiwaza-ai/releases/kamiwaza/${repo#"$SOURCE"/}:$tag" \
    || echo "$ref" >> copy-failures.txt
done < images.txt

skopeo copy --all --preserve-digests --retry-times 3 \
  "docker://$SOURCE/charts/kamiwaza:1.3.4" \
  "docker://$REGISTRY/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza:1.3.4" \
  || echo chart >> copy-failures.txt

# Only if a cluster administrator will install the VRAM plugin:
skopeo copy --all --preserve-digests --retry-times 3 \
  "docker://$SOURCE/charts/vram-plugin:1.3.4" \
  "docker://$REGISTRY/kamiwaza-ai/releases/kamiwaza/charts/vram-plugin:1.3.4" \
  || echo vram-plugin chart >> copy-failures.txt
```

The checks in the prerequisites for your environment (see online documentation) use two public images from Docker Hub that are not part of the release. Copy them too, keeping their Docker Hub paths. On a registry that uses projects, this needs projects named `library` and `curlimages`:

```
skopeo copy --all --preserve-digests \
  docker://docker.io/library/busybox:1.37 "docker://$REGISTRY/library/busybox:1.37"
skopeo copy --all --preserve-digests \
  docker://docker.io/curlimages/curl:latest "docker://$REGISTRY/curlimages/curl:latest"
```

Then, in those checks, use `<your registry host>/library/busybox:1.37` in place of `busybox:1.37`, and `<your registry host>/curlimages/curl:latest` in place of `curlimages/curl`, everywhere each one appears. Where a check names its image both in `--image` and inside `--overrides`, as the Kubernetes with Istio DNS check does, change both: the pod runs the one in `--overrides`.

If your registry requires credentials, run the checks after step 4, in the `kamiwaza` namespace, where the pull Secret lives; a pod in another namespace cannot use it. Give each check pod the Secret: add `"imagePullSecrets":[{"name":"kamiwaza-pull"}]` to the `spec` in its `--overrides` (add `--overrides='{"spec":{"imagePullSecrets":[{"name":"kamiwaza-pull"}]}}'` where the command has none), or `imagePullSecrets:` under `spec:` in a check's manifest.

Check:

```
cat copy-failures.txt
# Expect no output. Rerun the copy for any line listed; copies that already
# succeeded are skipped quickly.

helm show chart oci://$REGISTRY/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza --version
1.3.4
# Expect "name: kamiwaza" and "version: 1.3.4"

helm show chart oci://$REGISTRY/kamiwaza-ai/releases/kamiwaza/charts/vram-plugin --
version 1.3.4
# If you copied it: expect "name: vram-plugin" and "version: 1.3.4"
```

Check that the cluster's nodes can pull from your registry, which is the only proof that step 2 reached their container runtime. Like the prerequisite checks, the platform administrator runs this, with `REGISTRY` set. If your registry requires credentials, run it after step 4 instead, in the `kamiwaza` namespace, and add `--overrides='{"spec":{"imagePullSecrets":[{"name":"kamiwaza-pull"}]}}'` to `kubectl run`. In the `kamiwaza` namespace, `kubectl` prints a PodSecurity `restricted` warning for this pod; the check still runs.

```
kubectl -n default run registry-pull-check --restart=Never \
  --image=$REGISTRY/library/busybox:1.37 \
  --annotations=sidecar.istio.io/inject=false -- true
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded \
  pod/registry-pull-check --timeout=120s
# Expect "condition met". A pull error in
# `kubectl -n default describe pod registry-pull-check` names the cause:
# x509 means step 2 has not reached this node; "not found" means a missing copy.
kubectl -n default delete pod registry-pull-check
```

On OpenShift with GreyMatter, Kamiwaza's Ray pods also run the GreyMatter proxy image named in `global.greymatter.proxy.image`. That image is not part of the Kamiwaza release, and the step 5 values do not move it: copy it from the registry your GreyMatter platform pulls it from, e.g. to `$REGISTRY/greymatter-proxy:1.17.2`, and set `global.greymatter.proxy.image` to that copy.

A plain-HTTP registry. The commands on this page assume HTTPS. Against a registry that serves plain HTTP, turn off TLS in your copy tool (for `skopeo`, `--dest-tls-verify=false` on `copy` and `--tls-verify=false` on `login` and `inspect`), add `--plain-http` to every `helm` command that names your registry, including the install, and configure your cluster's container runtime to allow the registry. Registry credentials then cross the network unencrypted, so use plain HTTP only on a network you trust.

Step 4: Create a pull Secret, if your registry needs credentials

---

If your nodes can pull from your registry without credentials, skip this step.

Otherwise, create the pull Secret as described in Registry access in the prerequisites for Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation), with your registry's host as `<registry-host>`. It must match `$REGISTRY` exactly, including any port. Kamiwaza passes this Secret to its own pods and to every extension it deploys.

If a Secret of that name already exists, e.g. on an install that pulled from the internet until now, delete it first with `kubectl -n kamiwaza delete secret kamiwaza-pull`, then create it as above. Running pods keep running; pods started afterwards pull with the new Secret.

Step 5: Know the values to add

When you write your values file on your install page, merge these keys into it. Put them under the file's existing keys, and do not repeat any key at any level: not a second `global:` or `core:`, and not a second `scheduler:` under `core:`. When a key appears twice, Helm keeps only the last one and silently drops the other.

```
global:
  imageRegistryPrefix: <your registry host>/kamiwaza-ai/releases/kamiwaza
  # Only if you created a pull Secret in step 4:
  # imagePullSecrets:
  #   - name: kamiwaza-pull

core:
  scheduler:
    extensionImageRelocateRegistry: <your registry host>/kamiwaza-ai/releases/kamiwaza
  templates:
    availability:
      enabled: false
    sync:
      stage: LOCAL
      localMode:
        enabled: true
```

Value	What it does	If you leave it out
`global.imageRegistryPrefix`	Moves every image the chart deploys from the release location to the same path in your registry. `<prefix>/images/<name>`. It also moves the engine images of tenant-mode inference (see online documentation).	Every pod tries to pull location on the inte
`core.scheduler.extensionImageRelocateRegistry`	Rewrites each App Garden extension image, when the extension deploys, to this value followed by the last two segments of the image's path, which is where step 3 copied it.	Each App Garden images from the in fails. The install its
`core.templates.availability.enabled: false`	Turns off the App Garden's online catalog lookup.	The platform keeps Kamiwaza's online network cannot rea
`core.templates.sync.stage: LOCAL`	Selects the catalog that ships inside the release.	The install fails to r `templates.sync` requires templa
`core.templates.sync.localMode.enabled: true`	Loads the App Garden from that catalog.	The install fails to r `templates.sync` `templates.sync.`
`global.imagePullSecrets`	Names the pull Secret for the platform and its extensions.	Every pull from a r credentials fails wi

The two registry values are the same: your registry host followed by the release path, e.g. `registry.example.com:5000/kamiwaza-ai/releases/kamiwaza`, with no scheme such as `https://`. The chart and Kamiwaza supply the rest of each image's path.

Step 6: Install

Go to Installation Environments (see online documentation) and open the install page for your environment. Follow it as written, with the step 5 values merged into your values file. In every command on that page that names the release location, including the install command, use the chart from your registry instead:

```
oci://<your registry host>/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza
```

Check your completed values file against the chart just before you run the install command. The render needs no cluster access. Run it from the directory that holds `kamiwaza-values.yaml`, with `REGISTRY` set as above:

```
helm template kamiwaza oci://$REGISTRY/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza \
  --version 1.3.4 --namespace kamiwaza --values kamiwaza-values.yaml > rendered.yaml

grep -E '^\s+(- )?image: ' rendered.yaml | grep -v "image: \"\{0,1\}$REGISTRY/"
# Expect no output: every image the chart deploys comes from your registry

# Every image path contains /images/; this skips the prefix value itself.
grep -oE "$REGISTRY/[^\"' ,\\\\}]+" rendered.yaml | grep '/images/' \
  | sed -E 's/:[^/@]+@/@/' | sort -u > rendered-images.txt

# App Garden extension images keep their original registry in the render and
# are relocated when an extension deploys, so check their relocated copies too.
# Extensions run images pinned as tag@digest and relocated by tag; a reference
# without a digest is catalog display data that nothing pulls, so this checks
# only tag@digest references.
grep -oE "ghcr\.io/[^\"' ,\\\\}]+@sha256:[0-9a-f]{64}" rendered.yaml | grep -E ':
[^/@]+@' \
  | sed -E "s#^ghcr\.io/(.*/)?([^/]+/[^/:@]+):([^/@]+).*#$REGISTRY/kamiwaza-
ai/releases/kamiwaza/\2:\3#" \
  | sort -u >> rendered-images.txt

while read -r image; do        # with skopeo; any tool that reads a manifest works
  skopeo inspect --raw "docker://$image" > /dev/null 2>&1 || echo "missing: $image"
done < rendered-images.txt
# Expect no output: every image the chart deploys, and every App Garden image
# the render names in full, is in your registry. If you left out the GPU
# engines in step 1, expect them, and only them, listed.
```

If the render fails, the error names the value to fix. Seeing `ghcr.io` inside the rendered App Garden catalog is expected; Kamiwaza rewrites those references when an extension deploys.

Engine images for tenant-mode inference

Skip this section unless you will turn on tenant-mode inference (see online documentation), which serves models on GPUs and CPUs in a namespaced install.

Tenant-mode inference runs engine images that the core image's engine catalog pins by digest. The chart does not deploy them, so the step 6 check does not cover them, and Kamiwaza pulls each one by digest from `global.imageRegistryPrefix` followed by `images/` and the last segment of the catalog's repository, such as `$REGISTRY/kamiwaza-ai/releases/kamiwaza/images/vllm-cuda@sha256:...`. The release manifest lists every engine image at that path, so step 3 has already copied them, by digest, to where Kamiwaza looks; there is nothing more to copy. If you left out the GPU inference engines in step 1, those engines are not in your registry, and a model that needs one fails to deploy.

If this install served models before you moved it off the internet, its running inference deployments, including the platform's embedding model, still name the registry they were planned with. After the upgrade, stop each one and deploy it again. While `core.context.embedding.autoProvision` is on (the default, except on `global.platform: osgm`), the embedding model's deployment needs only the stop: the next embedding request deploys it again, from your registry. That request returns HTTP 503 with the code `embedding_deploying` until the new deployment is ready; retry it. With it off, the platform does not redeploy the model, and embedding requests return HTTP 503 with the code `embedding_unavailable` until you deploy it again yourself. See Engine images (see online documentation). On later upgrades, keep the previous release's engine images in your registry until every inference deployment has been redeployed.

VRAM plugin

Skip this section unless a cluster administrator will install the VRAM plugin (see online documentation). Its image, `images/vram-plugin`, is in the step 1 image list, and step 3 copies the image and the plugin's chart.

The plugin runs in its own namespace, `kamiwaza-vram-plugin`, not in `kamiwaza`. Create and label it as Share GPUs with the VRAM plugin (see online documentation) shows. If you created the pull Secret in step 4, copy it into that namespace. The copy is piped, so the credential never touches disk or a command line. Use server-side apply: a plain `kubectl apply` stores the whole Secret, credential included, in an annotation that `kubectl describe` prints.

```
kubectl -n kamiwaza get secret kamiwaza-pull -o yaml \
  | sed -e '/^  namespace:/d' -e '/^  uid:/d' -e '/^  resourceVersion:/d' -e '/^
creationTimestamp:/d' \
  | kubectl -n kamiwaza-vram-plugin apply --server-side -f -
```

Copy it again whenever you replace the Secret in `kamiwaza`. Then install the plugin from your registry, after `helm registry login` as in step 3:

```
helm upgrade --install vram-plugin oci://$REGISTRY/kamiwaza-
ai/releases/kamiwaza/charts/vram-plugin \
  --version 1.3.4 -n kamiwaza-vram-plugin \
  --set global.imageRegistryPrefix=$REGISTRY/kamiwaza-ai/releases/kamiwaza \
  --set 'imagePullSecrets[0].name=kamiwaza-pull'
```

Leave out the `imagePullSecrets` line if you skipped step 4. Without `global.imageRegistryPrefix`, the chart names the release location, and the plugin's pods cannot pull their image.

Troubleshooting

An App Garden extension fails to deploy, with pods in `ImagePullBackOff`. Describe one of the extension's pods and read the image it tries to pull:

```
kubectl -n kamiwaza get pods --field-selector=status.phase=Pending
kubectl -n kamiwaza describe pod <pod> | grep -E 'Image:|Failed'
```

If the image is `<your registry host>/kamiwaza-ai/releases/kamiwaza/<path>:<tag>`, that copy is missing: rerun the step 3 copy. If it still names `ghcr.io`, `core.scheduler.extensionImageRelocateRegistry` did not reach the platform: check your values file and rerun the install.

Pulls fail with `x509: certificate signed by unknown authority`. A node does not trust your registry's certificate authority. Repeat step 2 on that node, including the container runtime restart.

Pulls fail with `unauthorized` or `authentication required`. Check that the pull Secret exists in the `kamiwaza` namespace, that `global.imagePullSecrets` names it, and that its server matches your registry host exactly, including the port:

```
kubectl -n kamiwaza get secret kamiwaza-pull -o jsonpath='{.data.\.dockerconfigjson}' \
  | base64 -d | jq -r '.auths | keys[]'
# Expect your registry host, exactly as in $REGISTRY
```

`helm` returns `unauthorized` from your registry, though your copy tool works. Helm keeps its own login. Run `helm registry login` as in step 3.

Pods fail with `InvalidImageName`, image paths contain the same segment twice, or the render fails with `must include a repository path` or `must be a registry host[:port] with an optional lowercase /path`. A registry value has the wrong shape. Set both `global.imageRegistryPrefix` and `core.scheduler.extensionImageRelocateRegistry` to your registry host followed by `/kamiwaza-ai/releases/kamiwaza`, without a scheme.

Pods still pull from the release location on the internet, and the step 6 check lists them. `global.imageRegistryPrefix` did not reach the install. It must be set in your values file under the file's only `global:` key; the release's chart sets it to the release location, and a value you set wins.

The App Garden is empty after install. Ask the platform for the catalog error directly, with an administrator token:

```
curl -sS -X POST -H "Authorization: Bearer <token>" \
  https://<your Kamiwaza domain>/api/apps/remote/sync
```

An error that says the catalog targets a different Core version, or contains values unsupported by the catalog schema, means the chart and its catalog do not match. Check that you copied and installed the chart for the version you meant to install.

Prepare GPU nodes

A platform administrator prepares GPU nodes before Kamiwaza serves models on them. Kamiwaza installs nothing on a node: no driver, no container runtime configuration, and no GPU operator. This page covers each node-side requirement for NVIDIA GPUs, with a check and its expected output.

A default install cannot use GPUs even on prepared nodes. It installs without node reads and with tenant-mode inference off, and the install prints `WARNING: core cannot see GPUs on this install.` Prepare the nodes here, then turn on GPU serving with Tenant-mode inference (see online documentation), which uses a profile that the cluster owner signs.

Before you start

You need	Notes
Root access on each GPU node	The commands are for Ubuntu 22.04 and 24.04 nodes. On another distribution, use its packages; the checks are the same.
Cluster-admin access	To install the GPU Operator and create the `nvidia` RuntimeClass.
Packages and charts from inside your network	The node packages come from your mirror of the Ubuntu archive and of NVIDIA's container toolkit repository, and the GPU Operator chart and its images come from your registry.
`kubectl`, `helm`, and `jq`	On the machine you run the cluster checks from.
Two public test images	The checks run `docker.io/library/ubuntu:24.04` and `docker.io/nvidia/cuda:13.0.3-base-ubuntu24.04`. With internet access, use them as named. In an air gap, copy them into your registry first; the checks write them as `<YOUR_REGISTRY>/library/ubuntu:24.04` and `<YOUR_REGISTRY>/nvidia/cuda:13.0.3-base-ubuntu24.04`. If your registry needs credentials, the check pods run in the `default` namespace: create a pull Secret there and add `imagePullSecrets: [{name: <SECRET_NAME>}]` to each check pod's `spec`.

These steps were verified on a four-GPU NVIDIA A100 80GB host running k0s 1.36, the R580 server driver, and GPU Operator v26.7.1, except Refuse GPU requests by environment (see online documentation) and the checks for it in step 4, step 5, and the VRAM plugin section. Its two settings were verified on an NVIDIA GB10 (arm64) host running k0s 1.36, NVIDIA Container Toolkit 1.19.1 installed on the host, GPU Operator v25.10.1, and a VRAM plugin that passes the device-list mount: with both set, an unprivileged pod that asks for every GPU by environment sees none, and a VRAM plugin share still gets its GPU. Those checks test the same behavior, but were not run as written. The GPU Operator-managed toolkit variant was not verified. Where a step differs on k0s, the step says so.

Summary

Requirement	Check
Supported GPU and driver (see online documentation)	`nvidia-smi` reports the driver and CUDA version the engines need
Driver installed (see online documentation)	The driver loads, with Secure Boot on if the node uses it
Persistence mode (see online documentation)	`nvidia-smi` answers in well under a second
Container runtime (see online documentation)	The node advertises an `nvidia` runtime handler
GPU requests by environment refused (see online documentation) (required for tenant-mode GPU serving)	Both toolkit settings are set, and an unprivileged pod that asks for every GPU by environment sees none
`nvidia` RuntimeClass (see online documentation)	A privileged pod with `runtimeClassName: nvidia` sees the GPUs
GPU Operator (see online documentation)	Nodes advertise `nvidia.com/gpu`, and `nvidia.com/mig-*` for MIG
MIG layout (optional) (see online documentation)	Nodes advertise the MIG resources you configured
No taints on GPU nodes (see online documentation)	No GPU node carries a `NoSchedule` or `NoExecute` taint
Pinned GPU stack (see online documentation)	Driver, toolkit, and kernel packages are held
VRAM plugin (optional) (see online documentation)	Nodes advertise `kamiwaza.ai/vram-gb-gpu-<N>` for each shared GPU

Supported GPUs and driver floors

Kamiwaza serves models on NVIDIA GPUs. Each engine image is built against one CUDA major, and the host driver must support that major. `nvidia-smi` prints the highest CUDA version the installed driver supports in its header, as `CUDA Version: 13.0`.

Engine image	Engines	Minimum host driver	Lowest GPU compute capability
CUDA 13	llama.cpp, vLLM	580.65.06 (R580)	7.5 for llama.cpp; 8.0 for vLLM on amd64 and 12.1 (GB10) on arm64, and vLLM requires `bf16`
CUDA 12	llama.cpp	525.60.13; R570 or later for Blackwell GPUs	7.0
CUDA 12, built with CUDA 12.8	Diffusion	R570 or later recommended; not enforced	8.0, and the engine requires `bf16`

A GPU older than Volta (compute capability 7.0) has no engine image. Diffusion ships for amd64 only. vLLM ships for amd64, and for arm64 on GB10 only. llama.cpp ships for amd64 and arm64. See Engines (see online documentation) for what each engine serves.

Check on each GPU node:

```
nvidia-smi --query-gpu=index,name,driver_version,compute_cap --format=csv,noheader
nvidia-smi | grep -o 'CUDA Version: [0-9.]*'
```

```
0, NVIDIA A100 80GB PCIe, 580.178.04, 8.0
1, NVIDIA A100 80GB PCIe, 580.178.04, 8.0
CUDA Version: 13.0
```

Expect every GPU on a node to report the same name and compute capability. A profile binding cannot single out one kind of GPU on a node that has two (see Tenant-mode inference (see online documentation)), so keep such a node out of tenant-mode GPU serving.

Keep a node with two kinds of GPU out

A binding selects nodes by architecture only, so on a cluster with other GPU nodes a tenant-mode GPU pod could land on this one. After you install the GPU Operator in step 5 (see online documentation), turn off the operator's components on the node:

```
kubectl label nodes <NODE_NAME> nvidia.com/gpu.deploy.operands=false --overwrite
```

The operator then removes its device plugin, validator, GPU feature discovery, DCGM exporter and mig-manager from the node, so the scheduler can no longer give any pod one of its GPUs. That holds for every pod, not only tenant-mode ones; pods already running keep the GPUs they have. The host's driver and container toolkit stay, so the step 4 (see online documentation) check still passes there. In the step 5 check, the node shows `0` for its NVIDIA resources, or none at all. To undo it, remove the label: `kubectl label nodes <NODE_NAME> nvidia.com/gpu.deploy.operands-`.

A cluster whose only GPU node has two kinds of GPU, such as a single-node cluster, needs none of this: leave tenant-mode GPU serving off there, and publish no GPU binding for it.

Step 1: Install the driver

Upgrade the node and reboot before you install the driver. The kernel meta-package can be ahead of the kernel the node booted, and the driver's kernel modules track the meta-package:

```
export DEBIAN_FRONTEND=noninteractive
sudo -E apt-get update
sudo -E apt-get -y full-upgrade
sudo systemctl reboot
```

List the driver branches the archive offers for the node's GPUs:

```
sudo apt-get install -y ubuntu-drivers-common
sudo ubuntu-drivers list --gpgpu
```

```
nvidia-driver-580-server, (kernel modules provided by linux-modules-nvidia-580-server-
generic)
```

Install the driver together with the prebuilt kernel module package that the list names. The flavor at the end of its name, `-generic` above, matches your kernel. Naming the module package stops apt from building the module with DKMS:

```
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y \
  linux-modules-nvidia-580-server-generic nvidia-driver-580-server
sudo systemctl reboot
```

Check:

```
nvidia-smi -L
```

```
GPU 0: NVIDIA A100 80GB PCIe (UUID: GPU-8d898926-...)
```

Expect one line per GPU. Right after a driver load, `nvidia-smi` can print `couldn't communicate with the NVIDIA driver` for a few seconds while each GPU initializes.

Secure Boot

With UEFI Secure Boot on, the kernel loads only signed modules. Check the state:

```
mokutil --sb-state
```

```
SecureBoot enabled
```

The prebuilt module packages from step 1 are signed by Canonical and load without further steps. A module that DKMS builds is signed with a Machine Owner Key (MOK) on the node, and the firmware must trust that key. Enroll it once per node, from a console that reaches the pre-boot screen (physical, or a BMC or serial console); SSH cannot do the enrollment:

1. Create the key, if the node has none, and stage its enrollment. `mokutil` asks for a one-time password twice:

   ```
   sudo update-secureboot-policy --new-key
   sudo mokutil --import /var/lib/shim-signed/mok/MOK.der
   ```

2. Reboot. On the blue MOK manager screen, which waits about 10 seconds for a key press, select **Enroll MOK**, **Continue**, **Yes**, enter the one-time password, and select **Reboot**.

3. Rebuild the module so DKMS signs it with the enrolled key. Name the DKMS package of your driver branch, `nvidia-dkms-580-server` for the R580 server driver:

   ```
   sudo dpkg-reconfigure nvidia-dkms-580-server
   sudo systemctl reboot
   ```

Check:

```
mokutil --test-key /var/lib/shim-signed/mok/MOK.der
modinfo -F signer nvidia
nvidia-smi -L
```

Expect `is already enrolled`, a signer name, and one line per GPU. If the enrollment screen timed out. the request stays pending; reboot again. If the password is lost, clear the request with `sudo mokutil --revoke-import` and repeat step 1.

Step 2: Turn on persistence mode

Without persistence mode, every `nvidia-smi` call initializes each GPU again. On a four-GPU host that took about 13 seconds per call, long enough for the GPU Operator's validator to exceed the kubelet's 2-minute container start timeout (`RunContainerError: context deadline exceeded`), and the operator never became ready. Ubuntu's `nvidia-persistenced` service starts with `--no-persistence-mode`; override it:

```
sudo mkdir -p /etc/systemd/system/nvidia-persistenced.service.d
sudo tee /etc/systemd/system/nvidia-persistenced.service.d/10-persistence-mode.conf
>/dev/null <<'EOF'
[Service]
ExecStart=
ExecStart=/usr/bin/nvidia-persistenced --user nvidia-persistenced --persistence-mode --
verbose
EOF
sudo systemctl daemon-reload
sudo systemctl restart nvidia-persistenced
```

Check:

```
nvidia-smi --query-gpu=index,persistence_mode --format=csv,noheader
/usr/bin/time -f "%es" nvidia-smi -L >/dev/null
```

Expect `Enabled` for every GPU, and a time well under a second (0.04 seconds on the verification host).

Step 3: Configure the container runtime

Install the NVIDIA Container Toolkit from your mirror of NVIDIA's container toolkit repository:

```
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y nvidia-container-toolkit
nvidia-ctk --version
```

```
NVIDIA Container Toolkit CLI version 1.20.1
```

Then add an `nvidia` runtime to containerd. Keep `runc` as the default runtime: only pods that name the `nvidia` RuntimeClass use the NVIDIA runtime.

On a node whose containerd configuration you manage, let the toolkit write it, then restart containerd:

```
sudo nvidia-ctk runtime configure --runtime=containerd
sudo systemctl restart containerd
```

On k0s, k0s owns `/etc/k0s/containerd.toml`, so write the runtime as a drop-in by hand instead. k0s 1.36 runs containerd 2, whose drop-ins use config version 3:

```
sudo tee /etc/k0s/containerd.d/nvidia.toml >/dev/null <<'EOF'
version = 3

[plugins."io.containerd.cri.v1.runtime".containerd.runtimes.nvidia]
  privileged_without_host_devices = false
  runtime_type = "io.containerd.runc.v2"

[plugins."io.containerd.cri.v1.runtime".containerd.runtimes.nvidia.options]
  BinaryName = "/usr/bin/nvidia-container-runtime"
EOF
sudo systemctl restart k0sworker   # k0scontroller on a controller that runs workloads
```

Check from the cluster, for each GPU node. The node also lists an unnamed default handler, which the check leaves out:

```
kubectl get node <NODE_NAME> -o json | jq -r '[.status.runtimeHandlers[]?.name |
select(. != "")] | join(" ")'
```

```
nvidia runc
```

If your nodes do not report runtime handlers, this prints nothing; the pod check below is then the proof.

Refuse GPU requests by environment

Required on every node that serves tenant-mode GPU deployments.

By default the NVIDIA runtime gives a container on the `nvidia` RuntimeClass every GPU that its `NVIDIA_VISIBLE_DEVICES` variable names, whether the pod, a ConfigMap, or the image sets it. CUDA images, Kamiwaza's engine images among them, set it to `all`. So any pod that can use the `nvidia` RuntimeClass sees every GPU on the node without requesting one, and a quota on GPU resources does not limit it. Two NVIDIA Container Toolkit settings close this. With both set, the runtime gives an unprivileged container only the GPUs a device plugin allocated to it:

Setting	Effect
`accept-nvidia-visible-devices-as-volume-mounts = true`	The runtime accepts the device list a device plugin passes as volume mounts.
`accept-nvidia-visible-devices-envvar-when-unprivileged = false`	The runtime ignores `NVIDIA_VISIBLE_DEVICES` in an unprivileged container.

The order matters: accept volume mounts first, then make sure no device plugin on the node passes GPUs by environment, and refuse the environment last. A device plugin that passes volume mounts to a runtime that does not accept them hands out no usable GPU, and refusing the environment while a device plugin still uses it leaves every new GPU pod without a GPU.

On a node that runs no GPU pods yet, set both now. The GPU Operator's device plugin from step 5 (see online documentation) passes GPUs through CDI by default, not by environment:

```
sudo nvidia-ctk config --in-place --set accept-nvidia-visible-devices-as-volume-
mounts=true
sudo nvidia-ctk config --in-place --set accept-nvidia-visible-devices-envvar-when-
unprivileged=false
```

On a node that already serves GPUs, follow Turn the setting on for a node that serves GPUs (see online documentation) instead.

With the toolkit installed on the host, the runtime reads its configuration for every new container, so neither setting needs a restart. Privileged containers, such as the GPU Operator's own pods, are not affected. The pod security guard (see online documentation) keeps tenant pods from being privileged, adding capabilities, or mounting host paths, which are the ways around the setting.

If the GPU Operator manages the toolkit (`toolkit.enabled: true`), the commands above do not reach it. Its runtime reads `/usr/local/nvidia/toolkit/.config/nvidia-container-runtime/config.toml` under the operator's default `toolkit.installDir`, and takes both settings from the operator's `toolkit.env`. Add them to `gpu-operator-values.yaml` as strings, under the `toolkit:` key the file already has, beside any `toolkit.env` entries you already have:

```
toolkit:
  enabled: true
  env:
    - name: ACCEPT_NVIDIA_VISIBLE_DEVICES_AS_VOLUME_MOUNTS
      value: "true"
    - name: ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED
      value: "false"
```

Keep one `toolkit:` key. If a values file has two, Helm keeps the last one without a warning, and drops either these entries or `enabled: true`.

Apply the values with the `helm upgrade --install` command from step 5 (see online documentation). Helm changes only the operator's ClusterPolicy, and the operator updates the toolkit DaemonSet afterwards, so `rollout status` run straight after Helm can return before anything changed. Wait until the DaemonSet carries the new value, then for its rollout. Name the DaemonSet, the variable, and its new value:

```
DS=nvidia-container-toolkit-daemonset
NAME=ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED
VALUE=false
```

Then wait. Later steps run this same wait with other values:

```
for i in $(seq 60); do   # up to five minutes for the operator to update the DaemonSet
  CURRENT=$(kubectl -n gpu-operator get daemonset "$DS" \
    -o jsonpath="{.spec.template.spec.containers[*].env[?(@.name==\"$NAME\")].value}")
  [ "$CURRENT" = "$VALUE" ] && break
  sleep 5
done
echo "$NAME=$CURRENT"
kubectl -n gpu-operator rollout status "daemonset/$DS" --timeout=10m
```

Expect `ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED=false`, then `successfully rolled out`. Any other value on the first line means the operator has not applied the change yet, and the rollout result does not cover it. The DaemonSet replaces one node's pod at a time; on a large cluster, raise the timeout. Each toolkit pod it replaces removes the `nvidia` runtime from containerd and restarts containerd, and its replacement adds the runtime back and restarts containerd again. Running containers keep running, but a pod on the `nvidia` RuntimeClass that starts on that node meanwhile fails until the runtime is back, and is retried.

> **⚠ THE SETTING APPLIES TO THE WHOLE NODE**
>
> Every other workload on the node that selects GPUs by environment stops seeing them: unprivileged pods that set `NVIDIA_VISIBLE_DEVICES` instead of requesting `nvidia.com/gpu`, and `docker run --gpus` or Podman with the NVIDIA runtime without `--privileged`. Dedicate tenant-mode GPU nodes to Kamiwaza, or move those workloads to device-plugin requests before you turn the setting on.

Check on each GPU node:

```
grep -E '^accept-nvidia-visible-devices' /etc/nvidia-container-runtime/config.toml
```

```
accept-nvidia-visible-devices-as-volume-mounts = true
accept-nvidia-visible-devices-envvar-when-unprivileged = false
```

With a GPU Operator-managed toolkit, check the file its runtime reads instead, and expect the same two lines:

```
grep -E '^accept-nvidia-visible-devices' /usr/local/nvidia/toolkit/.config/nvidia-
container-runtime/config.toml
```

The step 4 (see online documentation) check proves it from the cluster.

The setting does not cover the variables that add other devices to a container that already has a GPU: `NVIDIA_MOFED`, `NVIDIA_GDS`, `NVIDIA_GDRCOPY`, and `NVIDIA_NVSWITCH`. On a node with InfiniBand or RDMA devices, a pod that has a GPU can add them with `NVIDIA_MOFED=enabled`, and RDMA traffic is not subject to Kubernetes network policy. No toolkit setting turns these off.

To turn the setting off again, for new containers at once:

```
sudo nvidia-ctk config --in-place --set accept-nvidia-visible-devices-envvar-when-unprivileged=true
```

Leave volume mounts accepted, so a device plugin that passes them keeps working. With the environment accepted again, any pod on the `nvidia` RuntimeClass can see every GPU on the node.

With a GPU Operator-managed toolkit, set `ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED` to `"true"` in `toolkit.env` instead, apply the values, and run the wait above with `DS=nvidia-container-toolkit-daemonset`, `NAME=ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED`, and `VALUE=true`.

Turn the setting on for a node that serves GPUs

1. If the VRAM plugin shares GPUs on the node, first upgrade it to the plugin from Kamiwaza 1.3.4 or later; see Share GPUs with the VRAM plugin (see online documentation). The plugin from Kamiwaza 1.3.3 and earlier passes each share's GPU by environment only. Then run the VRAM plugin's share check, and expect its `device-list mount` line: it proves the plugin passes the mount while the node still accepts the environment.

2. Accept volume mounts:

   ```
   sudo nvidia-ctk config --in-place --set accept-nvidia-visible-devices-as-volume-mounts=true
   ```

3. Read how the GPU Operator's device plugin passes GPUs, as in step 5 (see online documentation). With CDI, the default, change nothing. If it prints `envvar`, or nothing, set `volume-mounts` as step 5 describes, and apply the values. Then name the device plugin's DaemonSet, variable, and value, and run the wait from Refuse GPU requests by environment (see online documentation):

   ```
   DS=nvidia-device-plugin-daemonset
   NAME=DEVICE_LIST_STRATEGY
   VALUE=volume-mounts
   ```

4. Stop every GPU deployment in Kamiwaza and deploy it again, and check that each one serves. Do it now, after steps 1 to 3 and before step 5: a pod created now gets its GPU through CDI or a device-list mount, which the runtime still honors once it refuses the environment. A pod created before step 3 can hold its GPU by environment only, and when its container restarts after step 5, it starts without a GPU.

5. Refuse the environment:

```
sudo nvidia-ctk config --in-place --set accept-nvidia-visible-devices-envvar-
when-unprivileged=false
```

6. Run the checks above, the checks in step 4 and step 5, and the VRAM plugin's share check. In the step 5 allocation check, set `COUNT` to no more than the GPUs the node has free.

With a GPU Operator-managed toolkit, steps 2 and 5 add the matching `toolkit.env` entry instead: `ACCEPT_NVIDIA_VISIBLE_DEVICES_AS_VOLUME_MOUNTS` with `"true"`, then `ACCEPT_NVIDIA_VISIBLE_DEVICES_ENVVAR_WHEN_UNPRIVILEGED` with `"false"`. Apply each with the step 5 `helm upgrade --install` command, then wait as in Refuse GPU requests by environment (see online documentation), with `DS=nvidia-container-toolkit-daemonset` and `NAME` and `VALUE` set to the entry you added, before the next step.

Step 4: Create the `nvidia` RuntimeClass

```
cat <<'EOF' | kubectl apply -f -
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: nvidia
handler: nvidia
EOF
```

Check that a privileged pod on the `nvidia` RuntimeClass sees the GPUs. It is privileged because the node refuses GPU requests by environment from unprivileged pods (step 3 (see online documentation)), and it runs in `default`, which the pod security guard does not cover. The runtime adds `nvidia-smi` to the container, so a plain Ubuntu image from your registry is enough. The pod is pinned to one node: set `GPU_NODE` to its name, and run the check once for each GPU node. The pod has no tolerations, so a `NoSchedule` or `NoExecute` taint on the node keeps it `Pending`. If your GPU nodes carry either, do step 7 (see online documentation) first, or rerun this check after it.

If the API server refuses the pod with `violates PodSecurity`, your cluster enforces Pod Security on `default`, through a label on the namespace or a cluster-wide default. Then run this page's checks in a namespace of their own: create one, label it `pod-security.kubernetes.io/enforce=privileged`, use it in place of `default` in each check, and delete it when you are done.

```
GPU_NODE="<NODE_NAME>"
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: nvidia-runtime-check
  namespace: default
spec:
  runtimeClassName: nvidia
  restartPolicy: Never
  nodeSelector:
    kubernetes.io/hostname: ${GPU_NODE}
  containers:
  - name: smi
    image: <YOUR_REGISTRY>/library/ubuntu:24.04
    command: ["nvidia-smi", "-L"]
    securityContext:
      privileged: true
    env:
    - {name: NVIDIA_VISIBLE_DEVICES, value: all}
    - {name: NVIDIA_DRIVER_CAPABILITIES, value: utility}
EOF
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded pod/nvidia-runtime-
check --timeout=5m
kubectl -n default logs nvidia-runtime-check
kubectl -n default delete pod nvidia-runtime-check
```

Expect one `GPU n: ...` line per GPU on the node.

Check that the same request from an unprivileged pod gets no GPU. It runs on the node the check above set in `GPU_NODE`, and stops with an error if `GPU_NODE` is not set:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: nvidia-env-refusal-check
  namespace: default
spec:
  runtimeClassName: nvidia
  restartPolicy: Never
  nodeSelector:
    kubernetes.io/hostname: ${GPU_NODE:?set GPU_NODE to the node name, as in the check
above}
  containers:
  - name: env
    image: <YOUR_REGISTRY>/library/ubuntu:24.04
    command: ["sh", "-c", "ls /dev/nvidia[0-9]* 2>/dev/null || echo 'no GPU'"]
    env:
    - {name: NVIDIA_VISIBLE_DEVICES, value: all}
EOF
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded pod/nvidia-env-
refusal-check --timeout=5m
kubectl -n default logs nvidia-env-refusal-check
kubectl -n default delete pod nvidia-env-refusal-check
```

Expect `no GPU`. A list of `/dev/nvidia<N>` devices means the runtime still honors `NVIDIA_VISIBLE_DEVICES`; check the settings in step 3.

Use only the `nvidia` RuntimeClass. The GPU Operator in step 5 also creates `nvidia-cdi` and `nvidia-legacy` RuntimeClasses; on a node where the operator does not manage the toolkit, those have no containerd handler, and pods that name them fail to start. The operator takes ownership of the `nvidia` RuntimeClass, and uninstalling the operator deletes it.

Step 5: Install the GPU Operator

The NVIDIA GPU Operator's device plugin advertises GPUs to Kubernetes as `nvidia.com/gpu`, and MIG partitions as `nvidia.com/mig-<profile>`. With the driver and toolkit on the host, the operator must not install its own.

Save these values as `gpu-operator-values.yaml`. The `mig` and `migManager` blocks are needed only for MIG; keep them anyway, since they are harmless without it:

```
driver:
  enabled: false        # the host driver from step 1
toolkit:
  enabled: false        # the host toolkit and runtime from step 3
mig:
  strategy: mixed       # whole GPUs as nvidia.com/gpu, partitions as nvidia.com/mig-
<profile>
# On k0s with its default kubelet root, uncomment the next two lines. Leave them
# commented on a k0s started with --kubelet-root-dir=/var/lib/kubelet, as
# Kamiwaza's setup tooling starts it and as the VRAM plugin needs.
# hostPaths:
#   kubeletRootDir: /var/lib/k0s/kubelet
migManager:
  # Persistence mode (step 2) keeps the GPUs open, so mig-manager must stop
  # nvidia-persistenced before a MIG change and start it again afterwards.
  gpuClientsConfig:
    name: gpu-clients
extraObjects:
  - apiVersion: v1
    kind: ConfigMap
    metadata:
      name: gpu-clients
    data:
      clients.yaml: |
        version: v1
        systemd-services:
          - nvidia-persistenced.service
          - nvidia-dcgm.service
          - dcgm.service
          - dcgm-exporter.service
```

Install the operator chart from your registry, with its images mirrored there. With the values above, GPU Operator v26.7.1 runs these images; for another version, read the tags from `helm show values`:

Image	Values key to point at your copy
`nvcr.io/nvidia/gpu-operator:v26.7.1`	`operator.repository` and `validator.repository`
`nvcr.io/nvidia/k8s-device-plugin:v0.20.1`	`devicePlugin.repository` and `gfd.repository`
`nvcr.io/nvidia/cloud-native/k8s-mig-manager:v0.15.1`	`migManager.repository`
`nvcr.io/nvidia/k8s/dcgm-exporter:4.6.1-4.8.4-distroless`	`dcgmExporter.repository`
`registry.k8s.io/nfd/node-feature-discovery:v0.19.0`	`node-feature-discovery.image.repository`

Copy each image to your registry under the same path, e.g. `<YOUR_REGISTRY>/nvidia/gpu-operator:v26.7.1`. Then point the chart at your copies: save these values as `gpu-operator-images.yaml`. Each `repository` value is the image's path without its last segment, except `node-feature-discovery.image.repository`, which is the full image path. The chart supplies the tags in the table:

```
operator:
  repository: <YOUR_REGISTRY>/nvidia
validator:
  repository: <YOUR_REGISTRY>/nvidia
devicePlugin:
  repository: <YOUR_REGISTRY>/nvidia
gfd:
  repository: <YOUR_REGISTRY>/nvidia
migManager:
  repository: <YOUR_REGISTRY>/nvidia/cloud-native
dcgmExporter:
  repository: <YOUR_REGISTRY>/nvidia/k8s
node-feature-discovery:
  image:
    repository: <YOUR_REGISTRY>/nfd/node-feature-discovery
```

If your registry needs credentials, create a pull Secret in the `gpu-operator` namespace and list it under each component's `imagePullSecrets`.

Install with both values files. Helm merges them, so the `migManager` keys in each file apply together:

```
helm upgrade --install gpu-operator <GPU_OPERATOR_CHART> --version
<GPU_OPERATOR_VERSION> \
  -n gpu-operator --create-namespace -f gpu-operator-values.yaml -f gpu-operator-
images.yaml \
  --wait --timeout 10m
```

Check:

```
kubectl get clusterpolicy -o jsonpath='{.items[0].status.state}'; echo
kubectl get node <NODE_NAME> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
nvidia
```

```
ready
"nvidia.com/gpu":"4"
"nvidia.com/mig-3g.40gb":"0"
```

With `mig.strategy: mixed`, a node whose GPUs support MIG can also list MIG resources with a count of `0` until you partition a GPU in step 6. A node that has never had MIG instances may not list them at all.

The device plugin must not pass GPUs by environment, which the node refuses (step 3 (see online documentation)). Leave `cdi.enabled` at the chart's default, `true`: the device plugin then passes GPUs through CDI and needs no change. If you set `cdi.enabled: false`, also add this to `gpu-operator-values.yaml`, never `envvar`:

```
devicePlugin:
  env:
    - name: DEVICE_LIST_STRATEGY
      value: volume-mounts
```

Check how the device plugin passes GPUs:

```
kubectl -n gpu-operator get daemonset nvidia-device-plugin-daemonset \
  -o jsonpath='{.spec.template.spec.containers[*].env[?
(@.name=="DEVICE_LIST_STRATEGY")].value}'; echo
```

Expect a CDI strategy, such as `cdi-annotations,cdi-cri`, or `volume-mounts`. `envvar`, or no output, which means the plugin's default of `envvar`, gives every new GPU pod no GPU on a node that refuses the environment: set `volume-mounts` as above.

Check that a pod gets exactly the GPUs it requests. Set `COUNT` to the number of GPUs to request. Use a CUDA `base` image from your registry whose CUDA version is at or below the `CUDA Version` from the driver check; the container refuses to start on an older driver. Like the step 4 check, the pod cannot run on a node with a `NoSchedule` or `NoExecute` taint. If your GPU nodes carry either, do step 7 (see online documentation) first, or rerun this check after it.

```
COUNT=2
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: gpu-allocation-check
  namespace: default
spec:
  runtimeClassName: nvidia
  restartPolicy: Never
  containers:
  - name: cuda
    image: <YOUR_REGISTRY>/nvidia/cuda:13.0.3-base-ubuntu24.04
    command: ["bash", "-c", "nvidia-smi -L; nvidia-smi topo -p2p r"]
    resources:
      limits:
        nvidia.com/gpu: $COUNT
EOF
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded pod/gpu-allocation-
check --timeout=5m
kubectl -n default logs gpu-allocation-check
kubectl -n default delete pod gpu-allocation-check
```

Expect `COUNT` GPU lines, even though CUDA images ask for every GPU.

Step 6: Partition GPUs with MIG (optional)

MIG divides a supported GPU into hardware-isolated instances. Describe the layouts the node can take in the mig-manager configuration, then select one with a node label. Add a configuration to `gpu-operator-values.yaml` under `migManager`, e.g., one that leaves GPUs 0 and 1 whole and partitions GPUs 2 and 3 into two `3g.40gb` instances each. On a host bridged in pairs, keeping one bridged pair whole is what lets the whole GPUs serve tensor parallelism; see MIG and tensor parallelism (see online documentation).

```
migManager:
  config:
    create: true
    name: custom-mig-config
    default: all-disabled
    data:
      config.yaml: |-
        version: v1
        mig-configs:
          all-disabled:
            - devices: all
              mig-enabled: false
          gpu23-2x3g40gb:
            - devices: [0, 1]
              mig-enabled: false
            - devices: [2, 3]
              mig-enabled: true
              mig-devices:
                "3g.40gb": 2
  gpuClientsConfig:
    name: gpu-clients
```

Rerun the step 5 install with the updated values. mig-manager reads the new configuration only after the ConfigMap reaches its pod, and it acts only when the node label changes, so a label set too early fails with `selected mig-config not present` and is not retried. Confirm the ConfigMap holds the layout, then restart mig-manager and wait for it to be ready:

```
kubectl -n gpu-operator get configmap custom-mig-config \
  -o jsonpath='{.data.config\.yaml}' | grep -c 'gpu23-2x3g40gb:'
```

```
1
```

`kubectl wait` fails with `no matching resources found` until the DaemonSet has created the new pod, so the restart first waits for the pod to exist:

```
kubectl -n gpu-operator delete pod -l app=nvidia-mig-manager
for i in $(seq 30); do   # up to a minute for the new pod to appear
  kubectl -n gpu-operator get pod -l app=nvidia-mig-manager -o name | grep -q . &&
break
  sleep 2
done
kubectl -n gpu-operator wait --for=condition=Ready pod -l app=nvidia-mig-manager --
timeout=5m
```

Select the layout and wait for mig-manager. It evicts the GPU workloads on the node, changes the MIG mode, and creates the instances; on the verification host this took about 4 minutes:

```
kubectl label node <NODE_NAME> nvidia.com/mig.config=gpu23-2x3g40gb --overwrite
sleep 30   # mig-manager marks the new layout pending before it applies it
while :; do
  state=$(kubectl get node <NODE_NAME> -o
jsonpath='{.metadata.labels.nvidia\.com/mig\.config\.state}')
  echo "mig.config.state=$state"
  case "$state" in success|failed) break ;; esac
  sleep 15
done
```

Expect `mig.config.state=success`. On `failed`, read why, then restart mig-manager, which applies the labeled layout again when it starts:

```
kubectl -n gpu-operator logs -l app=nvidia-mig-manager --tail=30
kubectl -n gpu-operator delete pod -l app=nvidia-mig-manager
for i in $(seq 30); do   # up to a minute for the new pod to appear
  kubectl -n gpu-operator get pod -l app=nvidia-mig-manager -o name | grep -q . &&
break
  sleep 2
done
kubectl -n gpu-operator wait --for=condition=Ready pod -l app=nvidia-mig-manager --
timeout=5m
```

Then rerun the block that selects the layout, including its `sleep 30`. The label reads `failed` until the new pod updates it, so the loop alone stops at once.

Check:

```
nvidia-smi --query-gpu=index,mig.mode.current --format=csv,noheader
kubectl get node <NODE_NAME> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
nvidia
```

```
0, Disabled
1, Disabled
2, Enabled
3, Enabled
"nvidia.com/gpu":"2"
"nvidia.com/mig-3g.40gb":"4"
```

MIG instances do not survive a reboot, and on a virtual machine stopped and deallocated the GPUs' MIG mode is reset too. mig-manager applies the labeled layout again when it starts after boot, and pods that request a MIG resource stay `Pending` until it finishes. After each boot, check that `nvidia.com/mig.config.state` is `success`; if it is `failed`, see Troubleshooting (see online documentation).

MIG and tensor parallelism

A deployment that spans several GPUs needs a peer-to-peer path between every pair of GPUs the device plugin can give it. Check the paths between GPUs:

```
nvidia-smi topo -p2p r
```

```
        GPU0    GPU1    GPU2    GPU3
GPU0    X       OK      NS      NS
GPU1    OK      X       NS      NS
GPU2    NS      NS      X       OK
GPU3    NS      NS      OK      X
```

Only `OK` is a peer path. On this host the GPUs are bridged in pairs, 0 with 1 and 2 with 3, and have no path between the pairs. A multi-GPU request can land on any whole GPUs, so this node can offer tensor parallelism only if its whole GPUs are exactly one bridged pair. Partition the GPUs outside that pair with MIG, here GPUs 2 and 3, as the step 6 layout does. The owner's detector claims the `p2p` feature only when every pair of whole GPUs reports `OK`; see One replica across several devices (see online documentation).

Step 7: Remove taints from GPU nodes

Tenant-mode engine pods carry no tolerations, so they never schedule on a node with a `NoSchedule` or `NoExecute` taint. GPU nodes used for tenant-mode inference must carry neither.

Check every node that advertises an NVIDIA resource:

```
kubectl get nodes -o json | jq -r '.items[]
  | select(.status.allocatable | keys | any(startswith("nvidia.com/")))
  | [.metadata.name, ([.spec.taints[]? | select(.effect == "NoSchedule" or .effect ==
"NoExecute")
      | .key + ":" + .effect] | join(","))]
  | @tsv'
```

```
gpu-node-1
```

Expect each GPU node's name with nothing after it. For a node listed with a taint, remove it, repeating the key and effect the check printed, e.g., `nvidia.com/gpu:NoSchedule`:

```
kubectl taint nodes <NODE_NAME> <TAINT_KEY>:<EFFECT>-
```

A taint that your cluster needs, such as one that reserves GPU nodes for other workloads, cannot stay on the nodes Kamiwaza serves from.

Step 8: Pin the GPU stack

An unattended upgrade of the NVIDIA libraries while the old kernel module is still loaded breaks the driver until a reboot, and `nvidia-smi` then fails with `Failed to initialize NVML: Driver/library version mismatch`. Hold the driver, toolkit, and kernel packages, and upgrade them together in a maintenance window:

```
PKGS=$(dpkg-query -W -f='${db:Status-Abbrev} ${Package}\n' | awk '$1=="ii"{print $2}' \
  | grep -E '^(nvidia-|libnvidia-|linux-modules-nvidia-|linux-objects-nvidia-|linux-
signatures-nvidia-|linux-image-|linux-generic)')
sudo apt-mark hold $PKGS
```

Check:

```
apt-mark showhold | grep -E '^(nvidia-driver|nvidia-container-toolkit|linux-modules-
nvidia)'
```

Expect the driver, the container toolkit, and the kernel module package. Upgrade with `sudo apt-mark unhold $PKGS`, upgrade, reboot, and hold again.

Share GPUs with the VRAM plugin (optional)

The VRAM plugin lets several deployments share one GPU by its memory. Install it only if the cluster owner's signed profile has a binding with the `vram-plugin` allocator; see Tenant-mode inference (see online documentation). It accounts for memory and does not isolate it.

The plugin is a DaemonSet, `kamiwaza-vram-plugin`, with a ServiceAccount and a ClusterRole. It advertises each GPU it shares as `kamiwaza.ai/vram-gb-gpu-<N>`, where `<N>` is the GPU's index and the count is its memory in decimal GB. A cluster administrator installs it once per cluster; the Kamiwaza release does not. Its chart is published with the release from Kamiwaza 1.3.2; 1.3.0 and 1.3.1 publish no chart for it. The chart is for Kubernetes clusters: it ships no OpenShift SecurityContextConstraints.

The plugin runs in its own namespace, `kamiwaza-vram-plugin`, never in `kamiwaza`. Its ServiceAccount may get, watch, and patch every Node, and anyone who can create pods, workloads, or ServiceAccount tokens in its namespace can act as it. Give no tenant identity, including Kamiwaza's installer and service accounts, any rights in `kamiwaza-vram-plugin`. From Kamiwaza 1.3.4 the chart refuses to install into `kamiwaza` or `kamiwaza-extensions`.

Create the namespace and label it. The plugin mounts host paths, which needs the `privileged` Pod Security level, and it serves no mesh traffic:

```
kubectl create namespace kamiwaza-vram-plugin
kubectl label namespace kamiwaza-vram-plugin --overwrite \
  pod-security.kubernetes.io/enforce=privileged \
  pod-security.kubernetes.io/audit=privileged \
  pod-security.kubernetes.io/warn=privileged \
  istio-injection=disabled
```

Install it from the release:

```
helm upgrade --install vram-plugin oci://ghcr.io/kamiwaza-
ai/releases/kamiwaza/charts/vram-plugin \
  --version 1.3.4 -n kamiwaza-vram-plugin
```

The chart pulls its image from the release, `ghcr.io/kamiwaza-ai/releases/kamiwaza/images/vram-plugin`, without credentials. In an air gap, install it from your registry instead; see Install Without Internet Access (see online documentation).

By default the plugin runs on every node that GPU feature discovery from step 5 (see online documentation) labels `nvidia.com/gpu.present=true`, tolerates the `nvidia.com/gpu:NoSchedule` taint, sizes each GPU from the `nvidia.com/gpu.memory` label, and, unless you set `gpuIndices`, shares every GPU, MIG-enabled ones included. It sees neither MIG slices nor whole-device allocations, so keep two kinds of GPU out of it:

- **A MIG-enabled GPU.** The plugin advertises the card's full memory, and a deployment it places there finds no CUDA device and fails to start.
- **A GPU that something allocates whole.** A GPU the plugin shares is still offered as `nvidia.com/gpu`, and a whole-device binding covers every node of its architecture, so it can take that GPU while accounted deployments use it. Do not publish a whole-device binding for an architecture whose nodes the plugin shares GPUs on, and run nothing else there that requests `nvidia.com/gpu`.

List the GPUs to share, by their indices as `nvidia-smi` prints them, in `gpuIndices`. For example, on a node whose GPU 0 is MIG-partitioned and GPU 1 is whole, `--set 'gpuIndices={1}'` shares GPU 1, and GPU 0's slices stay available to MIG bindings. The step 6 (see online documentation) example node partitions GPUs 2 and 3 and keeps GPUs 0 and 1 for whole-device bindings, so the plugin has no GPU to share there. Every node the plugin runs on must have each listed index; on a node without one, the plugin exits.

A node you kept out (see online documentation) still carries `nvidia.com/gpu.present`, so the default placement would run the plugin there too. Keep it off such a node: label the node `kamiwaza.ai/gpu-vendor=none`, which the default placement excludes, or set `affinity` yourself.

The plugin registers with the kubelet only through `/var/lib/kubelet/device-plugins`, and the chart has no value to move it. On a node whose kubelet root is elsewhere, such as k0s's default `/var/lib/k0s/kubelet`, its pod does not become ready: it waits on the missing directory, or restarts when it cannot reach the kubelet. On k0s, install the plugin only on nodes whose k0s runs with `--kubelet-root-dir=/var/lib/kubelet`, as Kamiwaza's own setup tooling installs it.

Install it as release `vram-plugin` in the `kamiwaza-vram-plugin` namespace, as above: the same release name and namespace Kamiwaza's setup tooling uses, so the two do not contend for it.

Check:

```
kubectl -n kamiwaza-vram-plugin rollout status daemonset/kamiwaza-vram-plugin --
timeout=5m
kubectl get node <NODE_NAME> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
vram-gb
```

Expect `successfully rolled out`, then one `kamiwaza.ai/vram-gb-gpu-<N>` line for each GPU the plugin shares on the node.

A node that refuses GPU requests by environment (step 3 (see online documentation)) needs the VRAM plugin from Kamiwaza 1.3.4 or later. It passes each share's GPU as a device-list mount as well as by environment. The plugin from Kamiwaza 1.3.3 and earlier passes it by environment only, and on such a node the deployments it places start without a GPU. The share check below shows which of the two a node runs, whether or not the node refuses the environment yet.

The VRAM plugin is supported only with the toolkit installed on the host, as in step 3 (see online documentation), with `toolkit.enabled: false`. With `toolkit.enabled: true` and the chart's default `cdi.enabled: true`, the GPU Operator runs the NVIDIA runtime in CDI mode, where the GPU index the plugin passes does not name a CDI device. The deployments the plugin places then fail to start, whether or not the node refuses GPU requests by environment. From Kamiwaza 1.3.4, the chart refuses to install on such a cluster. Its error names the operator's ClusterPolicy and contains `vram-plugin: the GPU Operator on this cluster manages the NVIDIA container toolkit in CDI mode`. Set `allowGpuOperatorCdiToolkit=true` only if the nodes the plugin runs on use a host toolkit anyway, such as nodes labeled `nvidia.com/gpu.deploy.container-toolkit=false`.

Check that a share gets its GPU, and that the plugin passes it as a device-list mount. Set `GPU_INDEX` to a GPU the plugin shares on `GPU_NODE`. The pod asks for 1 GB of that GPU's memory, so it stays `Pending` while the GPU has none free:

```
GPU_NODE="<NODE_NAME>"
GPU_INDEX=<N>
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: vram-share-check
  namespace: default
spec:
  runtimeClassName: nvidia
  restartPolicy: Never
  nodeSelector:
    kubernetes.io/hostname: ${GPU_NODE}
  containers:
  - name: smi
    image: <YOUR_REGISTRY>/library/ubuntu:24.04
    command: ["sh", "-c", "ls /var/run/nvidia-container-devices 2>/dev/null | sed
's/^/device-list mount: /' | grep . || echo 'no device-list mount'; nvidia-smi -L
2>/dev/null || echo 'no GPU'"]
    env:
    - {name: NVIDIA_DRIVER_CAPABILITIES, value: utility}
    resources:
      limits:
        kamiwaza.ai/vram-gb-gpu-${GPU_INDEX}: 1
EOF
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded pod/vram-share-check
--timeout=5m
kubectl -n default logs vram-share-check
kubectl -n default delete pod vram-share-check
```

```
device-list mount: <N>
GPU 0: NVIDIA A100 80GB PCIe (UUID: GPU-...)
```

Expect exactly one `device-list mount` line, naming `GPU_INDEX`, and exactly one `GPU 0: ...` line: the container numbers its one GPU from 0. `no device-list mount` means the plugin passes the GPU by environment only: upgrade it before the node refuses the environment. On a node that already refuses it, that plugin's share also prints `no GPU`.

To remove the plugin, run `helm uninstall vram-plugin -n kamiwaza-vram-plugin`. It leaves its `kamiwaza.ai/vram-gb-gpu-<N>` node labels behind. Remove them with `kubectl label node <NODE_NAME> kamiwaza.ai/vram-gb-gpu-<N>-`, once for each GPU it shared.

Move a plugin installed in the kamiwaza namespace

The Kamiwaza 1.3.2 and 1.3.3 instructions installed the plugin in `kamiwaza`, where Kamiwaza's own identities can act as its ServiceAccount. Move it to `kamiwaza-vram-plugin`. Until you do, upgrading the old release with the 1.3.4 chart fails with `vram-plugin must not be released into tenant namespace "kamiwaza"`.

Helm cannot move a release, so uninstall it and install it again. Running deployments keep their GPU shares. New deployments that need a share wait until the new pod registers, usually a few seconds.

1. Create and label `kamiwaza-vram-plugin`, as above.

2. Save the release's values, so the new release gets the same settings:

```
helm get values vram-plugin -n kamiwaza -o yaml > vram-plugin-values.yaml
```

3. If the values name a pull Secret (`imagePullSecrets`), create it in `kamiwaza-vram-plugin` now, as Install Without Internet Access (see online documentation) shows. Otherwise the new pod cannot pull its image.

4. Get the chart onto this machine before you remove the old release, so nothing in the next two steps needs the network:

```
helm pull oci://ghcr.io/kamiwaza-ai/releases/kamiwaza/charts/vram-plugin --
version 1.3.4
```

   Without internet access, use the `vram-plugin-1.3.4.tgz` chart from your mirror instead, as Install Without Internet Access (see online documentation) describes.

5. Uninstall it, and wait until its pod is gone. The old pod removes the plugin's kubelet sockets as it stops, so the new one must not start first:

```
helm uninstall vram-plugin -n kamiwaza
kubectl -n kamiwaza wait --for=delete pod -l app.kubernetes.io/name=vram-plugin
--timeout=2m
```

6. Install it in its own namespace from the chart you saved, then run the check above:

```
helm upgrade --install vram-plugin ./vram-plugin-1.3.4.tgz \
  -n kamiwaza-vram-plugin -f vram-plugin-values.yaml
```

Next step

Give the cluster owner the node facts the signed profile attests. They run the detector from the Kamiwaza core image on each GPU node; see Tenant-mode inference (see online documentation).

Troubleshooting

The VRAM plugin advertises a GPU's memory wrongly, or its node has no `kamiwaza.ai/vram-gb-gpu-<N>` label. The plugin reads and labels its node through the Kubernetes API only when it starts. If it could not reach the API server then, it falls back to sizing the GPU from host memory and publishes no label. Check that no NetworkPolicy in `kamiwaza-vram-plugin` blocks its egress to the API server, then restart it with `kubectl -n kamiwaza-vram-plugin rollout restart daemonset/kamiwaza-vram-plugin`.

GPU Operator pods fail with `RunContainerError: context deadline exceeded`, and the operator never reports `ready`. Persistence mode is off; see step 2. Time `nvidia-smi -L` on the node: several seconds means persistence mode is off.

GPU pods stay `Pending` with `Insufficient nvidia.com/gpu`. Compare the request with the node's allocatable resources, from step 5. No NVIDIA resources at all means the device plugin is not running on the node, as expected on a node you kept out (see online documentation); otherwise check `kubectl -n gpu-operator get pods -o wide`. On k0s, a device plugin that runs but registers nothing means `hostPaths.kubeletRootDir` does not match the kubelet root: `/var/lib/k0s/kubelet` on a default k0s, and the GPU Operator's default, `/var/lib/kubelet`, on a k0s started with `--kubelet-root-dir=/var/lib/kubelet`.

MIG resources read `0` after a reboot, and `nvidia.com/mig.config.state` is `failed`. mig-manager could not apply the layout when the node started. Read its log:

```
kubectl get node <NODE_NAME> -o
jsonpath='{.metadata.labels.nvidia\.com/mig\.config\.state}'; echo
kubectl get node <NODE_NAME> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
mig
kubectl -n gpu-operator logs -l app=nvidia-mig-manager --tail=50 | grep -i
'fail\|error'
```

`failed to stop service nvidia-persistenced.service: timeout stopping service` means stopping the persistence daemon, which mig-manager does before a MIG change, took longer than mig-manager's 30-second limit. On the verification host it took 32 seconds while the node was still starting. Once the node is up, restart mig-manager, which applies the layout again:

```
kubectl -n gpu-operator delete pod -l app=nvidia-mig-manager
for i in $(seq 30); do   # up to a minute for the new pod to appear
  kubectl -n gpu-operator get pod -l app=nvidia-mig-manager -o name | grep -q . &&
break
  sleep 2
done
kubectl -n gpu-operator wait --for=condition=Ready pod -l app=nvidia-mig-manager --
timeout=5m
```

Then wait for `success` with the step 6 block that selects the layout, including its `sleep 30`: the label reads `failed` until the new pod updates it. `selected mig-config not present` means the label names a layout mig-manager has not loaded; see step 6.

GPU pods stay `Pending` with `had untolerated taint`. A GPU node is tainted; see step 7.

After the node refuses GPU requests by environment, a GPU deployment starts without a GPU. The engine log reports no CUDA device, or `libcuda.so.1: cannot open shared object file`. Either the pod was given its GPU before the change, so stop the deployment and deploy it again; or the device plugin still passes GPUs by environment (check it as in step 5); or the VRAM plugin is from Kamiwaza 1.3.3 or earlier (its share check prints `no device-list mount`). See Refuse GPU requests by environment (see online documentation).

Installing the VRAM plugin fails with an error such as `vram-plugin: image.tag "v0.2.6" is older than v0.2.7`. Your values pin the plugin's `image.tag` to a plugin version that passes each share's GPU by environment only. Remove `image.tag` from the values, so the chart runs its own release's image, and install again.

GPU pods fail with `no runtime for "nvidia" is configured`. The node's containerd has no `nvidia` runtime; check `runtimeHandlers` as in step 3. On k0s, check the drop-in's `version = 3` line and restart k0s.

`nvidia-smi` prints `Driver/library version mismatch`. The libraries were upgraded while the old kernel module is loaded. Reboot the node, then pin the stack as in step 8.

An engine exits at start with `CUDA driver version is insufficient for CUDA runtime version`. The host driver is older than the engine image's CUDA major. Upgrade the driver to the floor in Supported GPUs and driver floors (see online documentation), or have the owner attest the driver's CUDA major, so the platform picks an image the driver can run; see Attest the NVIDIA driver's CUDA major (see online documentation).

`nvidia-smi` finds no GPU with Secure Boot on. The kernel refused an unsigned module. Check `dmesg | grep -i 'module verification failed'`, then use the prebuilt, signed modules from step 1 or enroll a MOK as in Secure Boot (see online documentation).

Upgrade Kamiwaza

Within a release line, upgrade Kamiwaza in place: rerun the `helm upgrade --install` command from your environment's install page with the new `--version` and the same values files. An in-place upgrade keeps Kamiwaza's models, workrooms, users, data, and storage volumes, and existing extension deployments keep running.

Supported upgrades

You can upgrade in place from any 1.3 release to a later 1.3 release. There is no in-place upgrade from Kamiwaza 1.2 or earlier: contact your Kamiwaza representative to plan an assisted migration to a new installation.

Before you start

1. **Read the release notes** for every release after the one you run, up to and including the one you are installing. Each patch release lists any action it requires. See Release notes (see online documentation).

2. **Back up.** Back up Kamiwaza's PostgreSQL database and, for each Kaizen deployment, its database and its object store. For Kamiwaza's database, this writes a dump to a file on your machine:

   ```
   kubectl -n kamiwaza exec core-postgres-0 -c postgres -- \
     pg_dump -U core -d kamiwaza --format=custom > kamiwaza-before-upgrade.dump
   ```

   The dump holds everything in the database, so keep it as securely as the database itself. For each Kaizen deployment, run `pg_dumpall` in its PostgreSQL pod, and archive the `/data` directory of its SeaweedFS pod. Restoring a Kaizen backup into a new deployment is not documented; contact Kamiwaza support if you need one.

   From 1.3.4, keep the Kamiwaza database backup until you are sure you will stay on the new release. After an upgrade to 1.3.4 or later, Core from 1.3.3 or earlier does not start on the upgraded database, so a `helm rollback` to an earlier release leaves Kamiwaza down. To go back, restore this backup and install the earlier release; contact Kamiwaza support for help with the restore.

3. **Find your values files**, and the order you installed with.

4. **Air-gapped installations:** copy the new release into your registry, as described in Install Without Internet Access (see online documentation). Keep the previous release's engine images in your registry until every model deployment has been redeployed on the new release, and its extension images while deployments of those extension versions still run.

5. **Kubernetes with Istio installations:** if your values files set `global.authz.extAuthz.serviceHost`, it must name `core-api.<namespace>.svc.<cluster domain>` in Kamiwaza's namespace, for example `core-api.kamiwaza.svc.cluster.local`. From 1.3.3, any other value, such as `core-api.kamiwaza.svc` without the cluster domain, makes `helm upgrade` fail before it changes anything, with an error that the host `does not resolve into release namespace "<namespace>"`. Remove the value from your values files, or correct it. Do not turn off the access check that the error message names instead: that removes authentication from model or tool routes.

6. **Execution or dataset gates that use your own class:** from 1.3.4, Kamiwaza loads a gate class only from its own gate modules, from an installed gate package, or from a package listed in `AUTH_GATE_CLASSPATH_PREFIXES`. After the upgrade, an execution gate whose class is not allowed makes every job submission fail, local ones included, and a dataset whose gate class is not allowed is not served. Check both before you upgrade, as an administrator:
   - The execution gate: call `GET /api/cluster/execution-gate`.
   - Each gated dataset, one whose properties include `gate` or `gate_type`: call `GET /api/catalog/datasets/<URL-encoded URN>/gate`. The response names the dataset's gate class.

   A `403` with reason `classpath_not_allowed` means the class is not allowed. List its package in your values file before you upgrade, as described in Allow your gate's classpath (see online documentation).

Do not uninstall Kamiwaza before an upgrade. Uninstalling deletes the platform databases, the object store, and every extension's data.

Upgrade

As the identity that installed Kamiwaza, rerun the `helm upgrade --install` command from your environment's install page, with the new `--version` and every values file you installed with, in the same order. For a Kubernetes with Istio installation with one values file:

```
helm upgrade --install kamiwaza \
  oci://ghcr.io/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza \
  --version <new version> \
  --namespace kamiwaza \
  --values kamiwaza-values.yaml \
  --timeout 70m
```

Your environment's install page has the exact command and any additions:

- Kubernetes with Istio (see online documentation)
- OpenShift with GreyMatter (see online documentation). First repeat step 3 (see online documentation) with the new version, then rerun the upgrade command. With Helm 4, keep `--server-side=false` on every upgrade.
- Install Without Internet Access (see online documentation). Use the chart in your own registry, with the values from step 5 of that page. If you installed with the field kit, use the new release's field kit: render its values again and follow its README. Values rendered by an earlier kit refer to the earlier release's images.

Always upgrade with `helm upgrade --install` against the cluster, never by applying rendered manifests. The chart reads its generated keys and passwords back from the cluster on each upgrade.

Verify

- Every Kamiwaza deployment's pods are Ready, and the upgrade's jobs have completed. Helm can return before the pods are ready; wait with the command at the start of your install page's verify step.
- `helm list -n kamiwaza` shows the new chart version.
- The platform answers `200` at `/` and `401` at `/api/` without credentials.

- From 1.3.3, `GET /api/kamiwaza/metadata`, called as a signed-in user, reports the new `version`.

After the upgrade

Do each action that the release notes list for the releases you crossed, such as redeploying models.

Running extensions do not change. Each running extension deployment keeps the version it was deployed with. A selection changes only the version that new deployments use.

From 1.3.4, the App Garden moves older selections up to the release's versions. Each release ships a version of each extension it packages. When Kamiwaza 1.3.4 or later starts after the upgrade, every App Garden selection older than the version the release ships moves up to that version, including a version an administrator chose. A selection at or above that version does not change. A selection of an extension the release does not package, such as one published after the release, does not change either. An upgrade to 1.3.3 or earlier moves nothing.

To use a version newer than the release ships, as an administrator, open the App Garden and select **Check for updates**, then **Update** on a row, **Update all**, or **Choose version**.

If Kamiwaza cannot move a selection, it keeps the older one and records why. To check, use `ADMIN_TOKEN`, an access token for the `admin` user, as in Get an admin token (see online documentation):

```
curl --fail --silent --show-error --header "Authorization: Bearer $ADMIN_TOKEN" \
  "https://kamiwaza.example.com/api/apps/catalog" \
  | jq '{moved: [.baseline_advance.advanced[]? | "\(.name) \(.from_version) -> \
(.to_version)"],
         not_moved: [.baseline_advance.blocked[]? | {name, message}]}'
```

`moved` lists the selections Kamiwaza moved. `not_moved` should be empty. For each entry in it, follow its `message`, for example by selecting the release's version with **Choose version**.

A new Kaizen version is a new deployment. Each Kaizen deployment has its own database and storage volumes, and no data carries over from an existing deployment to a new one. Stopping a Kaizen deployment in the App Garden deletes its storage volume claims, and with a `Delete` reclaim policy, the Kubernetes default for dynamically provisioned volumes, its data. Keep your current Kaizen deployment unless you are starting fresh. To run a new Kaizen version alongside it:

1. Back up the existing deployment, as described in Before you start (see online documentation).
2. Make sure the cluster has room for a second complete Kaizen deployment. Without it, the new deployment's pods stay Pending and the deployment fails. Never stop the existing deployment to make room. If the new deployment fails before anyone has used it, or opens the Kamiwaza dashboard instead of Kaizen, add capacity if it was short, then stop that new, unused deployment and deploy again. Do not stop a new deployment that anyone has added data to; contact Kamiwaza support instead.
3. If the App Garden does not already select the new Kaizen version, select it, as described above.
4. Deploy Kaizen from the App Garden.
5. In the new deployment's Admin area, under **Models**, select the chat and embedding models. Upload your documents again, then choose **Index** on each one, or **Index selected** for several.

A custom-named default Global Milvus

From 1.3.4, only the Milvus VectorDB named `milvus-global-workroom` is the default for the Global Workroom. In 1.3.3 the default was the newest Milvus VectorDB in the Global Workroom, whatever its name. If an administrator created a Global Milvus with another name after `milvus-global-workroom`, or instead of it, that instance received every user's Global ingest.

On an affected installation, after the upgrade:

- The next Global ingest creates a new, empty `milvus-global-workroom`, unless one already exists. Global ingest and search use it.
- The custom instance's collections stay where they are, but no longer appear in default Global search or listings.
- The custom instance becomes an ordinary Global VectorDB owned by the administrator who created it. That administrator and other administrators can still reach it through the VectorDB API.

Check before you upgrade. As an administrator, list the Global Workroom's VectorDBs:

```
curl -sS -H "Authorization: Bearer $TOKEN" \
  -H "X-Workroom-ID: ffffffff-ffff-ffff-ffff-ffffffffffff" \
  "$KAMIWAZA_URL/api/context/vectordbs" |
  jq 'sort_by(.created_at) | .[] | {id, name, engine, created_at}'
```

The rows are listed oldest first. The installation is affected if the last row with `engine` `milvus` is not named `milvus-global-workroom`.

If you are affected, choose one:

1. **Ingest again (recommended).** Upgrade, then run ingestion again for the Global datasets and documents. The data lands in the new default. Then delete the custom instance if nothing else uses it.
2. **Adopt the custom instance.** This works only before any Global ingest runs on 1.3.4, while no `milvus-global-workroom` row exists. Rename the custom row. With the bundled PostgreSQL, in Kamiwaza's namespace:

   ```
   kubectl exec -i core-postgres-0 -n kamiwaza -- psql -U core -d kamiwaza <<'SQL'
   UPDATE context_vectordb_instances
      SET name = 'milvus-global-workroom'
    WHERE id = '<custom instance id>'
      AND NOT EXISTS (
        SELECT 1 FROM context_vectordb_instances
         WHERE name = 'milvus-global-workroom'
      );
   SQL
   ```

   `UPDATE 1` means the row was renamed. `UPDATE 0` means a `milvus-global-workroom` row already exists, so adopting no longer works; ingest again instead. If you renamed it after the upgrade, restart Core, and also `deployment/core-api` if your installation runs it:

```
kubectl -n kamiwaza rollout restart \
  deployment/core-scheduler deployment/core-raycluster-head deployment/core-
raycluster-worker
```

On startup, Kamiwaza releases the instance's owner, and it becomes the shared default with its existing collections.

More than one tenant. In 1.3.3, the first user to ingest into the Global Workroom became the owner of `milvus-global-workroom`. 1.3.4 releases that owner on startup, and removes the owner's access records only in the Global Workroom's tenant, or the default tenant. That covers every single-tenant installation. If other tenants exist, Kamiwaza logs:

```
Canonical Global VectorDB owner release cleaned relation tuples in tenant ... only; N
other tenant(s) exist.
```

A record left in another tenant grants nothing on the shared VectorDB. To keep the access records clean, remove it with `/api/authz/resources/context_vectordb/grants`.

App Garden apps no longer receive the sign-in cookies

From 1.3.4, on Kubernetes with Istio and on OpenShift with GrevMatter, an App Garden app no longer receives the Kamiwaza sign-in cookies (`access_token`, `access_token_refresh`, and the rest of that family) with each request. An app still receives its own cookies, unchanged. In earlier releases, any app a user opened could read that user's session and call Kamiwaza as that user, including as an administrator.

On OpenShift with GrevMatter, a Tool Shed tool also no longer receives any cookies, an `Authorization` header, or an `X-App-Session-Token` header, as on Kubernetes with Istio. Nothing in this section changes that for a tool. The usual upgrade (see online documentation) applies this change, and the trust and `enforce` settings below; on OpenShift with GreyMatter, that includes repeating step 3 of the install page.

Apps that call Kamiwaza as the signed-in user with the sign-in cookies stop working unless an administrator trusts them:

- **Kaizen** and **Kaizen (Legacy)** are trusted when an administrator deployed them from the App Garden on Kamiwaza 1.3.3 or later. A deployment created on an earlier release, or by a user who is not an administrator, is not trusted. Its model list is empty, chat reports that no chat models are available, and documents still upload but are not indexed for semantic search. Other features that call Kamiwaza as the user, such as tasks, are affected the same way.
- **Skills Library**, **Connector Builder**, and your own apps that pass the user's cookies to Kamiwaza are not trusted by default.

To trust a deployment, add its runtime identifier, the path segment after `/runtime/apps/` in the app's address, to `core.auth.runtimeCookies.trustedRuntimeTargets` in your values file, then rerun the upgrade command. Listing entries replaces the default list, so keep the two `template:` entries:

```
core:
  auth:
    runtimeCookies:
      trustedRuntimeTargets:
        - template:kaizen
        - template:kaizen-legacy
        - kaizen-1a2b3c4d   # an existing Kaizen deployment, by its identifier
```

A trusted app receives the session of every user who opens it, administrators included. Trust only apps you would trust with that session. Do not use wildcards or name prefixes: users choose deployment names.

For an existing Kaizen deployment, prefer its identifier to a redeploy. A new Kaizen deployment that an administrator deploys from the App Garden is trusted without an entry, but it does not carry over the existing deployment's data. See After the upgrade (see online documentation).

If an app you depend on breaks and you cannot trust it yet, set `core.auth.runtimeCookies.enforce: false` to restore the earlier behavior for every app. Tools still receive no cookies, `Authorization` header, or `X-App-Session-Token` header. This setting is temporary and will be removed in a later release.

The upgrade does not revoke anything an app could have obtained before it, such as an API token it created as a user. Administrators cannot see other users' API tokens. If you ran apps you do not trust, ask your users to open **Settings**, review the **API Tokens** on the **My Settings** tab, and revoke any they do not recognize.

Rotate the ForwardAuth signing secret

On Kubernetes with Istio and on OpenShift with GreyMatter, a user who could deploy an App Garden app or tool before 1.3.4 could read Kamiwaza's ForwardAuth signing secret through the API. Kamiwaza uses that secret to sign and verify the identity it passes to its own components. From 1.3.4 the API no longer returns it, but the secret itself is unchanged. Any signed-in user can deploy an App Garden app, so replace the secret after you upgrade, unless only administrators you trust could deploy apps and tools. Installations on Envoy ingress without Istio do not need this.

Do it after the upgrade, not before: until 1.3.4 is running, the new value could be read the same way.

What to expect. Users stay signed in. While the Ray head restarts, the console, the API, and app and model routes are unavailable, and jobs running on the Ray head are interrupted. Choose a quiet time.

The API calls use `ADMIN_TOKEN`, an access token for the `admin` user, as in Get an admin token (see online documentation), and `kamiwaza.example.com` for your Kamiwaza domain.

1. **Optional: find the deployments that stored the secret.** List App Garden deployments and look for the value Kamiwaza now withholds:

```
for kind in app tool service; do
  curl --fail --silent --show-error \
    --header "Authorization: Bearer $ADMIN_TOKEN" \
    "https://kamiwaza.example.com/api/apps/deployments?
workroom_id=all&template_type=$kind" \
    | jq -r '.[] | select(tostring | contains("platform ForwardAuth signing
secret"))
             | "\(.id) \(.name) owner=\(.owner_user_id)"'
done
```

Each line is a deployment whose environment holds the secret, so its owner could read it. Some deployments that held it are not listed: one that no longer exists, and one that stored it under a name Kamiwaza treats as a credential, such as a name ending in `_TOKEN`, `_SECRET`, `_PASSWORD`, or `_API_KEY`, because the response leaves such variables out. So an empty result does not show that nobody read the secret.

2. **Replace the value.** This writes a new random value into the `forwardauth-signature` Secret without printing it:

```
kubectl -n kamiwaza patch secret forwardauth-signature --type merge \
  --patch-file <(printf '{"data":{"secret":"%s"}}' \
    "$(openssl rand -base64 32 | tr -d '\n' | base64 | tr -d '\n')")
```

To confirm the change without printing the value, run `kubectl -n kamiwaza get secret forwardauth-signature -o jsonpath='{.data.secret}' | openssl dgst -sha256` before and after: the hash changes. Later upgrades keep the new value, because the chart reads the Secret back from the cluster.

Kamiwaza signs and verifies with `AUTH_FORWARD_HEADER_SECRET`, which the chart reads from this Secret, and passes `FORWARDAUTH_SIGNATURE_SECRET` to apps when it is set. If your values files set either variable, for example in `core.scheduler.extraEnv`, that entry takes precedence over the Secret. Its value is the one that was exposed, and replacing the Secret does not replace it. To find such entries, run this command. It prints Deployment and variable names, never values:

```
kubectl -n kamiwaza get deployments -o json \
  | jq -r '.items[] | .metadata.name as $d
      | .spec.template.spec.containers[] | (.env // [])
      | select(map(.valueFrom.secretKeyRef.name) | index("forwardauth-
signature"))
      | .[] | select(.name |
test("^(AUTH_FORWARD_HEADER_SECRET|FORWARDAUTH_SIGNATURE_SECRET)$"))
      | select(.valueFrom.secretKeyRef.name != "forwardauth-signature")
      | "\($d) \(.name)"'
```

If it prints nothing, go on to step 3. Otherwise, remove each listed variable from your values files, so that Kamiwaza uses the Secret you just replaced. If you must keep `AUTH_FORWARD_HEADER_SECRET` there, give it a new random value of at least 32 characters instead, for example from `openssl rand -base64 32`. Then rerun the upgrade command from your environment's install page instead of step 3: the upgrade restarts the same workloads.

3. **Restart every Kamiwaza workload that reads the secret, together:**

```
kubectl -n kamiwaza get deployments -o json \
  | jq -r '.items[]
      |
select([.spec.template.spec.containers[].env[]?.valueFrom.secretKeyRef.name]
               | index("forwardauth-signature"))
      | "deployment/" + .metadata.name' > forwardauth-readers.txt
cat forwardauth-readers.txt
xargs kubectl -n kamiwaza rollout restart < forwardauth-readers.txt
xargs -n 1 kubectl -n kamiwaza rollout status --timeout 15m < forwardauth-
readers.txt
```

The list is `core-scheduler`, `core-raycluster-head`, the Ray worker Deployment (`core-raycluster-worker`), and `core-api` if you enabled it.

4. **Verify.** The pods of each listed Deployment are Ready and were created after step 2. Sign in to the console, open an App Garden app, and send a request to a model deployment. If you removed variables in step 2, its command now prints nothing.

5. **OpenShift with GreyMatter:** redeploy each diffusion (image generation) model deployment. Until it is redeployed, it refuses requests.

Running App Garden apps keep working and need no action. An app of your own that verifies Kamiwaza's identity signature itself rejects requests until you redeploy it.

Users disabled or deleted before the upgrade

From 1.3.4, disabling or deleting a user who is linked to a Keycloak account also disables that account, ends its sessions, and revokes its API tokens. Users you disabled or deleted on an earlier release are refused by Kamiwaza once 1.3.4 runs, but their Keycloak accounts stay enabled until you repeat the change. Repeat it for each of them. A user you disabled in Keycloak itself needs nothing.

The API calls use `ADMIN_TOKEN`, an access token for the `admin` user, as in Get an admin token (see online documentation), and `kamiwaza.example.com` for your Kamiwaza domain.

1. **Disabled users.** List them:

```
curl --fail --silent --show-error --header "Authorization: Bearer $ADMIN_TOKEN"
\
  "https://kamiwaza.example.com/api/auth/users/" \
  | jq -r '.[] | select(.linked_subject_id != null and .active == false)
           | "\(.id) \(.username)"'
```

   For each `<id>`, disable the user again:

```
curl --silent --show-error --request PUT \
  --header "Authorization: Bearer $ADMIN_TOKEN" \
  --header 'Content-Type: application/json' --data '{"active": false}' \
  --write-out ' %{http_code}\n' \
  "https://kamiwaza.example.com/api/auth/users/<id>"
```

2. **Deleted users.** The API does not list them, so read their IDs from the database. This query changes nothing:

```
kubectl -n kamiwaza exec core-postgres-0 -c postgres -- \
  psql -U core -d kamiwaza -At -c "
    SELECT id, username FROM users
    WHERE deleted AND linked_subject_id IS NOT NULL
      AND linked_subject_id NOT IN (
        SELECT linked_subject_id FROM users
        WHERE NOT deleted AND linked_subject_id IS NOT NULL)"
```

   Each line is `<id>|<username>`. For each `<id>`, delete the user again:

```
curl --silent --show-error --request DELETE \
  --header "Authorization: Bearer $ADMIN_TOKEN" \
  --write-out ' %{http_code}\n' \
  "https://kamiwaza.example.com/api/auth/users/<id>"
```

3. **Check the responses.** Each request prints the user and then the HTTP status. `200` means the Keycloak account is now disabled, or no longer exists. A `503` names the step that failed; repeat the same request until it returns `200`.

Extension credentials from before 1.3.4

From 1.3.4, a new App Garden deployment keeps the passwords, tokens, and API keys in its environment in its own Kubernetes Secret, and App Garden deployment responses no longer return them. A deployment created before 1.3.4 still holds them as plain environment variables, in its pods and in its ConfigMap, until it is removed. Anyone who can read pods or ConfigMaps in Kamiwaza's namespace can read them there, including with the built-in `view` role, which cannot read Secrets.

For the extensions in the App Garden catalog, these values are the catalog's published defaults, such as `dev`, `postgres`, and `minioadmin`, unless an administrator entered a different value when deploying. A published default is not a secret, and a new deployment gets the same default, so these need nothing. Act only on a value that you or another administrator entered when deploying an extension before 1.3.4, such as an API key, a token, or a password.

Do not stop or replace a Kaizen or Kaizen (Legacy) deployment for this. A new deployment starts empty, and stopping a deployment in the App Garden deletes its data.

1. List the names each deployment from before 1.3.4 holds in plain text. The command prints names only, never values:

```
kubectl -n kamiwaza get configmap -l kamiwaza.io/extension-runtime=direct -o
json \
  | jq -r '.items[]
      | .metadata.labels as $l
      | [ (.metadata.annotations["extensions.kamiwaza.io/rendered-spec"] // "{}"
| fromjson)
          | .spec.services[]?.env[]?
          | select((.value // "") != "")
          | select((.name | test("(^|_)
(PASSWORD|PASSWD|PASSPHRASE|SECRET|TOKEN|APIKEY|CREDENTIALS?)$|(^|_)
(API|SECRET|PRIVATE|ACCESS|ENCRYPTION|SIGNING|MASTER)_KEY$"; "i"))
                  or (.value | tostring | test("://[^/?#@ ]*:[^/?# ]+@")))
          | .name ] | unique | select(length > 0)
      | "\($l["kamiwaza.io/deployment-id"] // "-") \
($l["extensions.kamiwaza.io/deployment-id"]): \(join(" "))"'
```

   Each line names a deployment by its ID and its runtime, then the variables it holds in plain text. A deployment created on 1.3.4 or later does not appear.

2. For each variable whose value an administrator entered, rotate the credential where it was issued: for example, revoke the API key at its provider and create a new one. If you used the same password anywhere else, change it there too. Until the deployment has the new value, whatever in it uses the credential stops working.

3. Check whether the deployment keeps data, using the runtime from step 1:

```
kubectl -n kamiwaza get pvc -l extensions.kamiwaza.io/deployment-id=<runtime>
```

- **No volume claims, and not Kaizen or Kaizen (Legacy):** deploy the extension again from the App Garden with the new value, check that the new deployment works, then stop the old deployment in the App Garden. Stopping it there also removes its plain-text copy.
- **Volume claims, or Kaizen or Kaizen (Legacy):** do not stop or replace the deployment. Contact Kamiwaza support.

4. A deployment that was stopped through the API (`DELETE /api/apps/deployment/<id>`), rather than in the App Garden, keeps its ConfigMap and its volumes, so it still appears in step 1. When you no longer need its data, remove it with `DELETE /api/apps/deployment/<id>/purge`, which deletes its volumes too. `<id>` is the ID that step 1 prints first.

Check: run step 1 again. The line of each deployment you replaced or purged is gone, and the new deployments do not appear. A deployment that step 3 leaves to Kamiwaza support is still listed.

Workroom context settings removed by the upgrade

From 1.3.4, only administrators can set the settings of an ontology or VectorDB instance, and the settings of an OmniParse instance other than its model selection, enable, timeout, and processing settings. Before 1.3.4, any member of a workroom could set them, including where an instance connects and with what credentials. The upgrade removes such settings from OmniParse, ontology, and VectorDB instances in a workroom other than the Global Workroom, and from the record each instance is deployed from, including values an administrator set: Kamiwaza cannot tell who set them. Instances in the Global Workroom do not change.

- **OmniParse:** every setting is removed except the model selection, enable, timeout, and processing settings.
- **Ontology and VectorDB:** every setting whose name marks an endpoint, URL, or host, a credential, a TLS or authentication setting, a proxy, a trust store, an identity, or a loader path is removed. Settings such as model names, timeouts, and sizes are kept, and a VectorDB keeps the credentials Kamiwaza generated for its own Milvus.
- **Graphiti endpoints:** Kamiwaza restores an ontology instance's chat and embedding endpoints from its chat and embedding models on the next ontology operation, which can return a `503` while the instance restarts; retry it. If Kamiwaza cannot restore an endpoint, the operation returns a `503` with the code `pre_gate_endpoint_scrubbed`, and an administrator selects the instance's chat and embedding models again, as in step 4 below.

Running pods. The first time Core starts after the upgrade, it also removes these settings from the running pods of the instances the upgrade changed. On each such pod, Core changes only the removed settings and any setting built from one of them. Each one gets the value the instance would be deployed with now, or is removed if it would not be set. For an OmniParse instance, Core first restores the settings Kamiwaza writes itself, such as its model endpoints and the Kamiwaza API address. The affected pods of each changed instance restart once, and its data volumes are kept. Other instances do not restart. A stopped or failed instance has no running pod to change; if it is started again, it starts without these settings. An ontology instance's chat and embedding endpoints are restored as described above.

The API calls use `ADMIN_TOKEN`, an access token for the `admin` user, as in Get an admin token (see online documentation), and `kamiwaza.example.com` for your Kamiwaza domain.

1. **Check what Core did.** Core's scheduler logs each instance it changed, and a summary. The lines name instances and settings, never their values, and each can appear twice, once as JSON:

```
kubectl -n kamiwaza logs deployment/core-scheduler -c core \
  | grep -e 'scrubbed context instance' -e 'scrubbed settings' \
    -e 'left the pod of' -e 'changed meanwhile' -e 'on its running pod'
```

- `Startup: removed scrubbed settings <settings> from the running pod of <type> instance <instance ID>; its pods restart once`: Core removed the settings from that instance's pod.
- `Startup: scrubbed context instance pod env reconcile {'found': …, 'updated': …, 'unchanged': …, 'skipped': …, 'errors': …}`: the summary, logged at every start. At a later start, it counts only the instances still listed. `errors` should be `0`, and `skipped` should count only instances whose deployment is not running, described next. If no summary line appears, restart the scheduler as shown below. If it still does not appear, contact Kamiwaza support.
- `Startup: left the pod of <type> instance <instance ID> for the next start: its deployment is not running`: the instance is stopped or failed, so it has no running pod to change. Core has already removed the settings from what the instance is deployed from, so if it is started again, it starts without them. Core logs this line, and counts the instance in `skipped`, at every start until the instance runs again or is deleted. At the first start after it runs again, Core checks its pod and stops listing it. No action is needed, unless the removed settings included a password for data the instance already holds: it would start without that password and could not open that data, so contact Kamiwaza support before you start it.
- `Startup: left the pod of <type> instance <instance ID> for the next start: <reason>` with any other reason, or a line that says Core `could not remove scrubbed settings` from an instance's pod: the instance was deploying, stopping, or being deleted, or the change failed. Core tries again at its next start. To try again now, wait until the instance has finished deploying or stopping, then restart the scheduler and check the log again:

  ```
  kubectl -n kamiwaza rollout restart deployment/core-scheduler
  kubectl -n kamiwaza rollout status deployment/core-scheduler --timeout
  15m
  ```

  If Core leaves the instance again for the same reason, contact Kamiwaza support.
- `Startup: the pod env record of <type> instance <instance ID> changed meanwhile; it is reconciled on the next start`: the instance's record changed while Core worked on it. so Core checks the instance again at its next start. The summary does not count this in `skipped`. To check now, restart the scheduler as shown above.
- `Startup: <type> instance <instance ID> keeps <settings> on its running pod`, logged as an error: Core leaves some settings on the pod, such as identity and credential settings. A new password, for example, would lock an instance out of data created with the old one. The value stays on the running pod, and Core logs that line again at every start. If the value is one an administrator set or Kamiwaza generated, you can leave it on the pod; otherwise, contact Kamiwaza support. Core does not keep the value for the instance's next deployment, though. If it is a password that protects data the instance already created, stopping and starting the instance would leave it unable to open that data. Do not stop that instance, and contact Kamiwaza support if it must keep the value.

2. **List the instances the upgrade could change.** This query changes nothing. It prints each instance's type, workroom ID, instance ID, and name, never its settings:

```
kubectl -n kamiwaza exec core-postgres-0 -c postgres -- \
  psql -U core -d kamiwaza -At -c "
    SELECT 'omniparse', workroom_id, id, name FROM context_omniparse_instances
    WHERE workroom_id <> 'ffffffff-ffff-ffff-ffff-ffffffffffff'
    UNION ALL
    SELECT 'ontology', workroom_id, id, name FROM context_ontology_instances
    WHERE workroom_id <> 'ffffffff-ffff-ffff-ffff-ffffffffffff'
    UNION ALL
    SELECT 'vectordb', workroom_id, id, name FROM context_vectordb_instances
    WHERE workroom_id <> 'ffffffff-ffff-ffff-ffff-ffffffffffff'
    ORDER BY 1, 2, 4"
```

Each line is `<type>|<workroom ID>|<instance ID>|<name>`.

3. **Find the values the upgrade removed.** Compare against your own records of what administrators set, or read an instance's settings from before the upgrade out of the backup you took in Before you start (see online documentation). This command changes nothing. It prints the instance's row from the backup. The instance's settings are the row's JSON object, credentials included, so run it where no one else can see the output:

```
kubectl -n kamiwaza exec -i core-postgres-0 -c postgres -- \
  pg_restore --data-only --table=context_omniparse_instances --file=- \
  < kamiwaza-before-upgrade.dump \
  | grep '^<instance ID>'
```

For an ontology or VectorDB instance, use the table `context_ontology_instances` or `context_vectordb_instances`. Optionally, cross-check the upgrade's own record: the database job's log names each instance the upgrade removed settings from, with its workroom and the names of the removed settings, never their values. By default the job and its log are removed about an hour after the job finishes:

```
kubectl -n kamiwaza logs -l app.kubernetes.io/name=core-db-init -c db-init --
tail=-1 \
  | grep -e 'removed privileged' -e 'config not an object'
```

A line that says `config not an object` names an instance whose settings Kamiwaza could not read. Core has no record of what the upgrade removed from it, so the values may stay on its running pod. Save it as below, or delete it. If its values must leave the running pod, contact Kamiwaza support.

4. **Re-enter the values an administrator set.** Save an instance only to re-enter values. A save stores them in the instance's settings and the record it is deployed from. It does not change the instance's running pod: if a value you re-enter must reach the running pod, contact Kamiwaza support. Each request prints the HTTP status; `200` means the instance was saved.

    - **OmniParse:** put the values in a file, so they stay off the command line, as `{"config": {"<setting>": "<value>"}}`. Then send it with the instance's workroom ID:

```
curl --silent --show-error --request PUT \
  --header "Authorization: Bearer $ADMIN_TOKEN" \
  --header "X-Workroom-ID: <workroom ID>" \
  --header 'Content-Type: application/json' \
  --data @omniparse-settings.json \
  --write-out ' %{http_code}\n' \
  "https://kamiwaza.example.com/api/context/omniparses/<instance ID>"
```

- **VectorDB:** send them the same way to `/api/context/vectordbs/<instance ID>`.
- **Ontology:** Kamiwaza 1.3 has no request to set an ontology instance's settings. Kamiwaza restores the chat and embedding endpoints itself, as described above. If an operation returns `pre_gate_endpoint_scrubbed`, select the instance's models again with `PUT /api/context/ontologies/<instance ID>/model-bindings` and the body `{"chat": {"deployment_id": "<chat model deployment ID>"}, "embedding": {"deployment_id": "<embedding model deployment ID>"}}`. To restore another setting the upgrade removed from an ontology instance, contact Kamiwaza support.

Installation Environments

Kamiwaza runs on several kinds of Kubernetes environment. How you prepare the cluster, and a few install values, depend on the environment: its service mesh, its security model, and how traffic reaches it.

Find your environment below. Each one has its own set of pages, written for that environment from start to finish:

1. **Prerequisites**, for the platform administrator: what the cluster must provide, with a check for each item.
2. **Install**: the license, your values file, and one `helm upgrade --install`.
3. **Uninstall**: removing Kamiwaza, and optionally what was prepared for it.

Use the same environment's pages throughout.

Environment	Service mesh	Pages
Kubernetes with Istio	Istio, with an ingress gateway	Prerequisites (see online documentation) · Install (see online documentation) · Uninstall (see online documentation)
OpenShift with GreyMatter	GreyMatter, with an edge published through an OpenShift Route	Prerequisites (see online documentation) · Install (see online documentation) · Uninstall (see online documentation)

Options that apply to any environment

- **No internet access.** Do Install Without Internet Access (see online documentation) before your environment's install page. It copies the release into your own registry.
- **GPUs and multiple nodes.** Each environment's prerequisites cover GPU nodes and multi-node clusters.

Not sure which environment you have, or yours is not listed? See Supported Configurations (see online documentation), or contact your Kamiwaza representative.

Platform Prerequisites for Kubernetes with Istio

Kamiwaza installs into a single namespace on a Kubernetes cluster that you already run. Before the install, a **platform administrator** (someone with cluster-admin rights) prepares the cluster so it provides everything on this page. The Kamiwaza chart does not create or repair any of it: it creates no cluster-scoped objects, installs no operators, and does not configure your service mesh. If a prerequisite is missing, the install fails or the platform comes up unable to serve traffic.

Work through each section and run its check. Each section also says what to **record**; the install (see online documentation) needs those facts.

Hardware sizing and supported operating systems are covered separately, in System Requirements (see online documentation).

Summary

Prerequisite	What the cluster must provide
Namespace (see online documentation)	A pre-created namespace, labeled for Istio sidecar injection
Installer identity (see online documentation)	A namespace-scoped identity to run Helm as. Never cluster-admin
Pod security guard (see online documentation)	An admission policy that enforces Pod Security `baseline` in the `kamiwaza` and `kamiwaza-extensions` namespaces. Needs Kubernetes 1.30 or later
Istio and the ingress gateway (see online documentation)	An Istio mesh in sidecar mode, with an ingress gateway serving the Kamiwaza domain on HTTPS
Edge certificate (see online documentation)	A TLS certificate for the domain on that gateway
DNS (see online documentation)	The domain resolves to the gateway, from clients and from inside the cluster
Block storage (see online documentation)	An RWO StorageClass whose volumes keep their data when pods are replaced
Object storage (see online documentation)	A decision: the built-in object store, or your own S3-compatible service
Kubernetes API addresses (see online documentation)	The API server's Service and endpoint addresses, recorded for network policy
Registry access (see online documentation)	Nodes can pull Kamiwaza images
Node kernel limits (see online documentation)	`fs.inotify` limits at or above the floor on every node
Outbound network access (see online documentation)	HTTPS to the image registry and to any model sources you use

GPU inference and multi-node clusters add requirements; see GPU nodes (see online documentation) and Multi-node clusters (see online documentation).

Kamiwaza installs into a namespace named `kamiwaza`; use that name. The examples use `kamiwaza.example.com` as the domain; substitute your own. Run the checks with the administrator's kubeconfig unless a check says otherwise.

Namespace

Create the `kamiwaza` namespace. The installer does not create it, and the install never uses `--create-namespace`.

Label it for Istio sidecar injection, and to warn and audit against the Pod Security `restricted` profile. Do not enforce `restricted`: some Kamiwaza components, and Istio's init container, do not meet it yet.

Do not add an `enforce` label for `baseline` either. Istio's `istio-init` container adds the `NET_ADMIN` and `NET_RAW` capabilities, which `baseline` forbids, and Pod Security admission can exempt a whole pod but never one container. So `enforce=baseline` rejects every pod Istio injects. The pod security guard (see online documentation) enforces `baseline` instead.

```
kubectl create namespace kamiwaza
kubectl label namespace kamiwaza \
  istio-injection=enabled \
  pod-security.kubernetes.io/warn=restricted \
  pod-security.kubernetes.io/audit=restricted
```

If your mesh uses revision-based injection (`istio.io/rev=<revision>`), use that label instead of `istio-injection=enabled`.

Check:

```
kubectl get namespace kamiwaza --show-labels
# Expect istio-injection=enabled (or istio.io/rev=...), and the pod-security labels
```

Installer identity

Helm runs as a namespace-scoped identity, never as cluster-admin. It needs:

- the built-in `admin` ClusterRole, bound **in the namespace only** with a RoleBinding;
- create and manage rights on the Istio resources Kamiwaza writes into its namespace (`VirtualService`, `DestinationRule`, `EnvoyFilter`, `AuthorizationPolicy`, `PeerAuthentication`);
- rights to manage `endpoints` and patch `pods/status`. The chart grants these to its own service accounts, and Kubernetes only lets an identity grant permissions it holds itself.

This manifest creates a service account with those rights. You can bind the same Roles to a user or group from your identity provider instead.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kamiwaza-installer
  namespace: kamiwaza
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kamiwaza-installer-admin
  namespace: kamiwaza
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
  - kind: ServiceAccount
    name: kamiwaza-installer
    namespace: kamiwaza
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kamiwaza-installer-istio
  namespace: kamiwaza
rules:
  - apiGroups: ["networking.istio.io"]
    resources: ["destinationrules", "envoyfilters", "virtualservices"]
    verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
  - apiGroups: ["security.istio.io"]
    resources: ["authorizationpolicies", "peerauthentications"]
    verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kamiwaza-installer-istio
  namespace: kamiwaza
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: kamiwaza-installer-istio
subjects:
  - kind: ServiceAccount
    name: kamiwaza-installer
    namespace: kamiwaza
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kamiwaza-installer-delegation
  namespace: kamiwaza
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["create", "delete", "patch", "update"]
  - apiGroups: [""]
    resources: ["pods/status"]
    verbs: ["patch"]
---
```

```
  apiVersion: rbac.authorization.k8s.io/v1
  kind: RoleBinding
  metadata:
    name: kamiwaza-installer-delegation
    namespace: kamiwaza
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: Role
    name: kamiwaza-installer-delegation
  subjects:
    - kind: ServiceAccount
      name: kamiwaza-installer
      namespace: kamiwaza
```

Save it as `kamiwaza-installer.yaml` and apply it as the administrator:

```
kubectl apply -f kamiwaza-installer.yaml
```

Give the installer a kubeconfig for this identity, following your cluster's normal practice for issuing credentials. For example, with a service account token, run as the administrator:

```
(
  set -e
  SERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
  kubectl config view --minify --raw \
    -o jsonpath='{.clusters[0].cluster.certificate-authority-data}' | base64 -d >
cluster-ca.crt
  TOKEN=$(kubectl -n kamiwaza create token kamiwaza-installer --duration=24h)

  export KUBECONFIG=kamiwaza-installer.kubeconfig
  kubectl config set-cluster kamiwaza --server="$SERVER" \
    --certificate-authority=cluster-ca.crt --embed-certs
  kubectl config set-credentials kamiwaza-installer --token="$TOKEN"
  kubectl config set-context kamiwaza --cluster=kamiwaza --user=kamiwaza-installer --
namespace=kamiwaza
  kubectl config use-context kamiwaza
)
```

The parentheses keep these commands from changing your own shell's `KUBECONFIG`. If your kubeconfig names a `certificate-authority` file instead of embedding `certificate-authority-data`, copy that file to `cluster-ca.crt` instead. The token expires, and the API server may cap the duration you ask for. Issue a new one before each later upgrade or uninstall.

Check that the identity can create namespaced resources and cannot create cluster-scoped ones:

```
SA=system:serviceaccount:kamiwaza:kamiwaza-installer
kubectl auth can-i create deployments -n kamiwaza --as=$SA            # yes
kubectl auth can-i create clusterroles --as=$SA                       # no
kubectl auth can-i create customresourcedefinitions --as=$SA          # no
kubectl auth can-i list nodes --as=$SA                                # no
```

The installer's Istio rights are checked in the next section, once Istio's resource types exist.

Pod security guard

The installer identity holds `admin` in the namespace, so it can create pods, and so can the service accounts Kamiwaza installs. Without an admission policy, anyone holding one of those identities can start a privileged pod, or one that mounts the node's file system, and take over the node. The namespace labels above only warn.

The guard closes that. It is a ValidatingAdmissionPolicy and binding that enforce the Kubernetes Pod Security `baseline` controls on every pod in the `kamiwaza` and `kamiwaza-extensions` namespaces, including init and ephemeral containers. That includes the control Kubernetes 1.34 added to `baseline`: a probe or lifecycle hook may not name a `host`.

The guard makes one exception: an `istio-init` container may add `NET_ADMIN` and `NET_RAW`, but only in the shape Istio injects. The exception requires Istio's own proxy image and entrypoint, and the `istio-iptables` command. The only volume it may mount is the service account token volume that Kubernetes adds to the pod (`kube-api-access-*`), read-only at the token path. The guard checks that volume's contents, not only where it is mounted: it must project just the token, the cluster CA certificate and the pod's namespace. The container's environment variables must have literal values: Istio copies the mesh's `proxyMetadata` settings, such as DNS capture, into `istio-init` that way. A variable set with `valueFrom` or `envFrom` is refused, and so are names that change which code runs: `PATH`, `GCONV_PATH`, the dynamic loader's `LD_*`, and the iptables module paths `XTABLES_LIBDIR`, `IPTABLES_LIB_DIR` and `IP6TABLES_LIB_DIR`. A container merely named `istio-init` gets nothing.

A second policy, `kamiwaza-tenant-exec-guard`, refuses `kubectl exec` and `kubectl attach` into any container named `istio-init` in the same namespaces. The installer's `admin` role includes exec, and `istio-iptables` has flags that keep it running, so without this policy the installer could start a pod whose `istio-init` stays up and open a root shell in it with `NET_ADMIN` and `NET_RAW`. Exec and attach into every other container work as before.

Both policies bind cluster administrators too, and a namespace admin cannot read, change or remove them.

What remains is inherent to classic Istio injection: `istio-init` must run with `NET_ADMIN` and `NET_RAW` to write the pod's iptables rules. The guard keeps tenants from forging that container and from exec'ing or attaching into it. It pins the image, the entrypoint and the `istio-iptables` command, not every flag, so a tenant can still choose how `istio-iptables` configures its own pod's network, but cannot run code of its own with those capabilities. Istio's ambient mode has no `istio-init` container at all.

The guard needs the `admissionregistration.k8s.io/v1` ValidatingAdmissionPolicy API, which Kubernetes 1.30 and later serve. For older clusters, see Kubernetes 1.28 and 1.29 (see online documentation) below.

Save this as `kamiwaza-tenant-pod-guard.yaml`:

```yaml
# Kamiwaza tenant pod guard.
#
# Enforces the Kubernetes Pod Security "baseline" controls, including the
# probe and lifecycle-hook host control added in Kubernetes 1.34, on every pod
# in the Kamiwaza namespaces. There is one exception: Istio's classic
# `istio-init` init container may add NET_ADMIN and NET_RAW, and only in the
# exact shape istiod injects. A second policy refuses exec and attach into
# istio-init. Requires Kubernetes 1.30 or later. Apply as a cluster
# administrator.
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
  name: kamiwaza-tenant-pod-guard
  labels:
    app.kubernetes.io/name: kamiwaza-tenant-pod-guard
    app.kubernetes.io/part-of: kamiwaza
spec:
  failurePolicy: Fail
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
        apiVersions: ["v1"]
        operations: ["CREATE", "UPDATE"]
        resources: ["pods", "pods/ephemeralcontainers"]
  matchConditions:
    # Pod specs are immutable apart from images, tolerations, deadlines and
    # ephemeral containers. Skipping metadata-only updates keeps pods admitted
    # before the guard existed patchable (labels, ownerReferences) until they
    # are replaced. Every create and every spec change is checked.
    - name: create-or-spec-change
      expression: "request.operation == 'CREATE' || object.spec != oldObject.spec"
  variables:
    - name: podSC
      expression: "has(object.spec.securityContext) ? object.spec.securityContext : {}"
    - name: initContainers
      expression: "has(object.spec.initContainers) ? object.spec.initContainers : []"
    - name: ephemeralContainers
      expression: "has(object.spec.ephemeralContainers) ?
object.spec.ephemeralContainers : []"
    - name: allContainers
      expression: "object.spec.containers + variables.initContainers +
variables.ephemeralContainers"
    - name: baselineCapabilities
      expression: >-
        ['AUDIT_WRITE', 'CHOWN', 'DAC_OVERRIDE', 'FOWNER', 'FSETID', 'KILL', 'MKNOD',
        'NET_BIND_SERVICE', 'SETFCAP', 'SETGID', 'SETPCAP', 'SETUID', 'SYS_CHROOT']
    # Where the mesh pulls its proxy image from. Each entry ends in ':' or '@'
    # so a look-alike repository name cannot match. Edit for a mirrored registry.
    - name: istioInitImagePrefixes
      expression: >-
        ['registry.istio.io/release/proxyv2:', 'registry.istio.io/release/proxyv2@',
        'docker.io/istio/proxyv2:', 'docker.io/istio/proxyv2@',
        'ghcr.io/kamiwaza-internal/containers/images/istio/proxyv2:',
        'ghcr.io/kamiwaza-internal/containers/images/istio/proxyv2@']
    # Environment names istio-init may not set, because they steer what code it
    # loads: the dynamic loader (LD_*, GCONV_PATH), the command search path, and
    # the directory iptables loads its extension modules from. Exact names, so
    # istio-iptables' own settings (IPTABLES_PROBE_PORT, for example) still pass.
    - name: istioInitLoaderEnvPrefixes
      expression: "['LD_']"
```

```
    - name: istioInitLoaderEnvNames
      expression: >-
        ['PATH', 'GCONV_PATH', 'XTABLES_LIBDIR', 'IPTABLES_LIB_DIR',
        'IP6TABLES_LIB_DIR']
    # Volumes in exactly the shape the ServiceAccount admission plugin injects
    # (kube-api-access-*): a projected token, the cluster CA from the
    # kube-root-ca.crt ConfigMap, and the pod's namespace. They hold no file a
    # tenant chooses, so they are the only volumes istio-init may mount.
    - name: serviceAccountTokenVolumes
      expression: >-
        (has(object.spec.volumes) ? object.spec.volumes : []).filter(v,
        has(v.projected) && has(v.projected.sources) &&
        size(v.projected.sources) == 3 &&
        v.projected.sources.exists(s, has(s.serviceAccountToken) &&
        s.serviceAccountToken.path == 'token' &&
!has(s.serviceAccountToken.audience)) &&
        v.projected.sources.exists(s, has(s.configMap) &&
        s.configMap.name == 'kube-root-ca.crt' && has(s.configMap.items) &&
        size(s.configMap.items) == 1 && s.configMap.items[0].key == 'ca.crt' &&
        s.configMap.items[0].path == 'ca.crt') &&
        v.projected.sources.exists(s, has(s.downwardAPI) && has(s.downwardAPI.items) &&
        size(s.downwardAPI.items) == 1 && s.downwardAPI.items[0].path == 'namespace' &&
        has(s.downwardAPI.items[0].fieldRef) &&
        s.downwardAPI.items[0].fieldRef.fieldPath == 'metadata.namespace')
        ).map(v, v.name)
    - name: safeSysctls
      expression: >-
        ['kernel.shm_rmid_forced', 'net.ipv4.ip_local_port_range',
        'net.ipv4.ip_unprivileged_port_start', 'net.ipv4.tcp_syncookies',
        'net.ipv4.ping_group_range', 'net.ipv4.ip_local_reserved_ports',
        'net.ipv4.tcp_keepalive_time', 'net.ipv4.tcp_fin_timeout',
        'net.ipv4.tcp_keepalive_intvl', 'net.ipv4.tcp_keepalive_probes']
    - name: selinuxTypes
      expression: "['', 'container_t', 'container_init_t', 'container_kvm_t',
'container_engine_t']"
    # Containers whose probe or lifecycle hook names a host. The kubelet sends
    # these requests from the node, so a host could aim them at a node-local or
    # cloud metadata endpoint. Baseline allows the field unset or empty.
    - name: handlerHostContainers
      expression: >-
        variables.allContainers.filter(c,
        ((has(c.livenessProbe) ? [c.livenessProbe] : []) +
        (has(c.readinessProbe) ? [c.readinessProbe] : []) +
        (has(c.startupProbe) ? [c.startupProbe] : [])).exists(h,
        (has(h.httpGet) && has(h.httpGet.host) && h.httpGet.host != '') ||
        (has(h.tcpSocket) && has(h.tcpSocket.host) && h.tcpSocket.host != '')) ||
        (has(c.lifecycle) &&
        ((has(c.lifecycle.postStart) ? [c.lifecycle.postStart] : []) +
        (has(c.lifecycle.preStop) ? [c.lifecycle.preStop] : [])).exists(h,
        (has(h.httpGet) && has(h.httpGet.host) && h.httpGet.host != '') ||
        (has(h.tcpSocket) && has(h.tcpSocket.host) && h.tcpSocket.host != '')))
        ).map(c, c.name)
  validations:
    - expression: >-
        !(has(object.spec.hostNetwork) && object.spec.hostNetwork) &&
        !(has(object.spec.hostPID) && object.spec.hostPID) &&
        !(has(object.spec.hostIPC) && object.spec.hostIPC)
      message: "hostNetwork, hostPID and hostIPC are not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
    - expression: >-
        variables.allContainers.all(c, !has(c.securityContext) ||
```

```
        !has(c.securityContext.privileged) || !c.securityContext.privileged)
      messageExpression: >-
        'privileged containers are not allowed in the Kamiwaza tenant namespace: ' +
        variables.allContainers.filter(c, has(c.securityContext) &&
        has(c.securityContext.privileged) && c.securityContext.privileged).map(c,
c.name).join(', ')
      message: "privileged containers are not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
    - expression: "!has(object.spec.volumes) || object.spec.volumes.all(v,
!has(v.hostPath))"
      messageExpression: >-
        'hostPath volumes are not allowed in the Kamiwaza tenant namespace: ' +
        object.spec.volumes.filter(v, has(v.hostPath)).map(v, v.name).join(', ')
      message: "hostPath volumes are not allowed in the Kamiwaza tenant namespace."
      reason: Forbidden
    # Regular and ephemeral containers: the baseline capability set only.
    - expression: >-
        (object.spec.containers + variables.ephemeralContainers).all(c,
        !has(c.securityContext) || !has(c.securityContext.capabilities) ||
        !has(c.securityContext.capabilities.add) ||
        c.securityContext.capabilities.add.all(cap, cap in
variables.baselineCapabilities))
      message: "containers may add only the Pod Security baseline capabilities in the
Kamiwaza tenant namespace."
      reason: Forbidden
    # Init containers: the baseline set, or NET_ADMIN/NET_RAW for an injected
    # istio-init. The exception pins the mesh's own image, its entrypoint (no
    # command), the istio-iptables subcommand, and no restart policy (a plain
    # init container, not a long-running sidecar). Its only mounts may be the
    # service-account token volume (serviceAccountTokenVolumes), read-only at
    # the token path. Environment entries must be literal values: istiod copies
    # the mesh's proxyMetadata (DNS capture, for example) into istio-init as
    # name/value pairs. valueFrom, envFrom and the loader names above are
    # refused.
    - expression: >-
        variables.initContainers.all(c,
        !has(c.securityContext) || !has(c.securityContext.capabilities) ||
        !has(c.securityContext.capabilities.add) ||
        c.securityContext.capabilities.add.all(cap, cap in
variables.baselineCapabilities) ||
        (c.name == 'istio-init' &&
        c.securityContext.capabilities.add.all(cap,
        cap in variables.baselineCapabilities || cap in ['NET_ADMIN', 'NET_RAW']) &&
        variables.istioInitImagePrefixes.exists(p, c.image.startsWith(p)) &&
        !has(c.command) &&
        has(c.args) && size(c.args) > 0 && c.args[0] == 'istio-iptables' &&
        ( !has(c.env) || c.env.all(e, !has(e.valueFrom) &&
        !variables.istioInitLoaderEnvPrefixes.exists(p, e.name.startsWith(p)) &&
        !(e.name in variables.istioInitLoaderEnvNames))) && !has(c.envFrom) &&
        !has(c.volumeDevices) && !has(c.restartPolicy) &&
        ( !has(c.volumeMounts) || c.volumeMounts.all(m,
        m.mountPath == '/var/run/secrets/kubernetes.io/serviceaccount' &&
        has(m.readOnly) && m.readOnly && !has(m.subPath) && !has(m.subPathExpr) &&
        m.name in variables.serviceAccountTokenVolumes)) &&
        has(c.securityContext.allowPrivilegeEscalation) &&
        !c.securityContext.allowPrivilegeEscalation &&
        has(c.securityContext.capabilities.drop) &&
        'ALL' in c.securityContext.capabilities.drop))
      messageExpression: >-
        init containers may add only the Pod Security baseline capabilities in the
        Kamiwaza tenant namespace; NET_ADMIN and NET_RAW are allowed only for the
```

```
        Istio-injected istio-init container.
      reason: Forbidden
    - expression: >-
        variables.allContainers.all(c, !has(c.ports) ||
        c.ports.all(p, !has(p.hostPort) || p.hostPort == 0))
      message: "hostPort is not allowed in the Kamiwaza tenant namespace."
      reason: Forbidden
    - expression: "size(variables.handlerHostContainers) == 0"
      messageExpression: >-
        'probes and lifecycle hooks may not set a host in the Kamiwaza tenant
namespace: ' +
        variables.handlerHostContainers.join(', ')
      message: "probes and lifecycle hooks may not set a host in the Kamiwaza tenant
namespace."
      reason: Forbidden
    - expression: >-
        !(has(variables.podSC.seccompProfile) && variables.podSC.seccompProfile.type
== 'Unconfined') &&
        variables.allContainers.all(c, !has(c.securityContext) ||
        !has(c.securityContext.seccompProfile) ||
        c.securityContext.seccompProfile.type != 'Unconfined')
      message: "seccompProfile Unconfined is not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
    - expression: >-
        !(has(variables.podSC.appArmorProfile) &&
variables.podSC.appArmorProfile.type == 'Unconfined') &&
        variables.allContainers.all(c, !has(c.securityContext) ||
        !has(c.securityContext.appArmorProfile) ||
        c.securityContext.appArmorProfile.type != 'Unconfined') &&
        ( !has(object.metadata.annotations) || object.metadata.annotations.all(k,
        !k.startsWith('container.apparmor.security.beta.kubernetes.io/') ||
        object.metadata.annotations[k] == 'runtime/default' ||
        object.metadata.annotations[k].startsWith('localhost/')))
      message: "AppArmor Unconfined is not allowed in the Kamiwaza tenant namespace."
      reason: Forbidden
    - expression: >-
        ( !has(variables.podSC.seLinuxOptions) ||
        (( !has(variables.podSC.seLinuxOptions.type) ||
variables.podSC.seLinuxOptions.type in variables.selinuxTypes) &&
        ( !has(variables.podSC.seLinuxOptions.user) ||
variables.podSC.seLinuxOptions.user == '') &&
        ( !has(variables.podSC.seLinuxOptions.role) ||
variables.podSC.seLinuxOptions.role == ''))) &&
        variables.allContainers.all(c, !has(c.securityContext) ||
 !has(c.securityContext.seLinuxOptions) ||
        (( !has(c.securityContext.seLinuxOptions.type) ||
c.securityContext.seLinuxOptions.type in variables.selinuxTypes) &&
        ( !has(c.securityContext.seLinuxOptions.user) ||
c.securityContext.seLinuxOptions.user == '') &&
        ( !has(c.securityContext.seLinuxOptions.role) ||
c.securityContext.seLinuxOptions.role == '')))
      message: "custom SELinux user, role or type is not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
    - expression: >-
        variables.allContainers.all(c, !has(c.securityContext) ||
        !has(c.securityContext.procMount) || c.securityContext.procMount ==
'Default')
      message: "procMount Unmasked is not allowed in the Kamiwaza tenant namespace."
      reason: Forbidden
    - expression: >-
```

```
        !has(variables.podSC.sysctls) ||
        variables.podSC.sysctls.all(s, s.name in variables.safeSysctls)
      message: "only the Pod Security baseline safe sysctls are allowed in the Kamiwaza
tenant namespace."
      reason: Forbidden
    - expression: >-
        !(has(variables.podSC.windowsOptions) &&
has(variables.podSC.windowsOptions.hostProcess) &&
        variables.podSC.windowsOptions.hostProcess) &&
        variables.allContainers.all(c, !has(c.securityContext) ||
        !has(c.securityContext.windowsOptions) ||
        !has(c.securityContext.windowsOptions.hostProcess) ||
        !c.securityContext.windowsOptions.hostProcess)
      message: "Windows HostProcess pods are not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBinding
metadata:
  name: kamiwaza-tenant-pod-guard
  labels:
    app.kubernetes.io/name: kamiwaza-tenant-pod-guard
    app.kubernetes.io/part-of: kamiwaza
spec:
  policyName: kamiwaza-tenant-pod-guard
  # [Warn, Audit] reports what would be denied without denying it. Use it for a
  # staged rollout on an existing install, then switch to [Deny, Audit].
  validationActions: [Deny, Audit]
  matchResources:
    namespaceSelector:
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: In
          values: [kamiwaza, kamiwaza-extensions]
---
# The exception above gives NET_ADMIN and NET_RAW to istio-init, and a
# namespace admin can create a pod whose istio-init keeps running. Exec or
# attach into it would give the tenant a root shell with those capabilities in
# the pod's network namespace, so this policy refuses both.
#
# Exec and attach reach admission as CONNECT on pods/exec and pods/attach, and
# `object` is the request's PodExecOptions or PodAttachOptions. Kubernetes
# v1.30.0 source: the ValidatingAdmissionPolicy plugin handles Connect
# (staging/src/k8s.io/apiserver/pkg/admission/plugin/policy/validating/plugin.go:79);
# CONNECT is a valid policy operation
# (pkg/apis/admissionregistration/validation/validation.go:473-479); the
# connect handler admits the parsed options as the object
# (staging/src/k8s.io/apiserver/pkg/endpoints/handlers/rest.go:211).
# `container` is absent when the client names none. The API server then picks
# the pod's only regular container, never an init container
# (pkg/registry/core/pod/strategy.go:668-691), so that request is allowed.
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
  name: kamiwaza-tenant-exec-guard
  labels:
    app.kubernetes.io/name: kamiwaza-tenant-exec-guard
    app.kubernetes.io/part-of: kamiwaza
spec:
  failurePolicy: Fail
  matchConstraints:
```

```
    resourceRules:
      - apiGroups: [""]
        apiVersions: ["v1"]
        operations: ["CONNECT"]
        resources: ["pods/exec", "pods/attach"]
  validations:
    - expression: "!has(object.container) || object.container != 'istio-init'"
      message: "exec and attach into istio-init are not allowed in the Kamiwaza tenant
namespace."
      reason: Forbidden
---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBinding
metadata:
  name: kamiwaza-tenant-exec-guard
  labels:
    app.kubernetes.io/name: kamiwaza-tenant-exec-guard
    app.kubernetes.io/part-of: kamiwaza
spec:
  policyName: kamiwaza-tenant-exec-guard
  # Binds the same namespaces as kamiwaza-tenant-pod-guard, in the same mode.
  validationActions: [Deny, Audit]
  matchResources:
    namespaceSelector:
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: In
          values: [kamiwaza, kamiwaza-extensions]
```

Mirrored proxy image. The guard recognizes Istio's proxy image by its repository, listed in `istioInitImagePrefixes`. If your mesh pulls the proxy from another registry or under another name, for example a mirror such as `registry.example.com/istio/proxyv2`, add that repository before you apply the guard. End each entry with `:` and add a second entry ending in `@`, as the existing entries do. Otherwise the guard denies every pod Istio injects. A registry mirror that keeps the original image name, such as a containerd mirror, needs no change.

Apply it as the administrator:

```
kubectl apply -f kamiwaza-tenant-pod-guard.yaml
```

Check that the four objects exist, and that the installer is denied a privileged pod. The server-side dry run creates nothing:

```
kubectl get validatingadmissionpolicy,validatingadmissionpolicybinding \
  kamiwaza-tenant-pod-guard kamiwaza-tenant-exec-guard

SA=system:serviceaccount:kamiwaza:kamiwaza-installer
kubectl create --dry-run=server -f - --as=$SA \
  --as-group=system:serviceaccounts \
  --as-group=system:serviceaccounts:kamiwaza \
  --as-group=system:authenticated <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: kamiwaza-pod-guard-probe
  namespace: kamiwaza
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  hostPID: true
  containers:
    - name: probe
      image: busybox:1.37
      securityContext:
        privileged: true
EOF
# Expect Forbidden, with a message that names
# ValidatingAdmissionPolicy 'kamiwaza-tenant-pod-guard'
```

Check that the installer is denied exec into `istio-init`. Admission runs before the API server looks the pod up, so the request needs no pod:

```
kubectl create --raw \
  "/api/v1/namespaces/kamiwaza/pods/kamiwaza-exec-guard-probe/exec?container=istio-
init&command=true&stdout=true" \
  -f /dev/null --as=$SA \
  --as-group=system:serviceaccounts \
  --as-group=system:serviceaccounts:kamiwaza \
  --as-group=system:authenticated
# Expect Forbidden, with a message that names
# ValidatingAdmissionPolicy 'kamiwaza-tenant-exec-guard'.
# NotFound means the request got past the guard.
```

Existing installs. On a cluster that already runs Kamiwaza, roll the guard out in stages, so that it never denies a pod the platform needs:

1. Apply it with `validationActions: [Warn, Audit]` in both bindings. The guard then reports what it would deny without denying it.
2. Check every running pod in both namespaces the guard covers against it. Most pods are created by controllers, so a `helm upgrade` does not show the guard's warnings; submitting each pod's spec again as a server-side dry run does, and creates nothing:

```
for ns in kamiwaza kamiwaza-extensions; do
  for pod in $(kubectl -n "$ns" get pods -o name); do
    kubectl -n "$ns" get "$pod" -o json \
      | jq 'del(.metadata.uid, .metadata.resourceVersion,
.metadata.ownerReferences,
                .metadata.creationTimestamp, .metadata.generateName, .status,
                .spec.nodeName, .spec.ephemeralContainers)
            | .metadata.name += "-guard-check"' \
      | kubectl create --dry-run=server -f - 2>&1 >/dev/null \
      | grep -E "kamiwaza-tenant-pod-guard|^Error" && echo "  from $ns/$pod"
  done
done
# Expect no output, apart from "No resources found" for a namespace
# with no pods
```

Any `Error` line means that pod's spec could not be checked; resolve it before you continue.

A warning about a `hostPath` volume means the install uses the single-node shared model cache (`core.modelsStorage.sharedHostPath.enabled`), a developer setting; turn it off and upgrade first. A warning about `istio-init` means the injected container does not match the guard's exception. Most often its proxy image is not in `istioInitImagePrefixes`. Otherwise, compare the pod's `istio-init` container with the exception in the manifest: the guard refuses an environment variable set with `valueFrom` or `envFrom` or named for a loader path, and any mount other than the service account token volume.

3. When the check prints no warning and no `Error` line, change both bindings to `validationActions: [Deny, Audit]` and apply the file again.

Pods that were running before the guard keep running. The guard checks a pod when it is created or its spec changes.

How a denial shows up. A pod created directly, or an ephemeral container from `kubectl debug`, is refused with `Forbidden` and the guard's message. A Deployment, StatefulSet or Job is accepted, but its controller cannot create the pod: `kubectl -n kamiwaza get events --field-selector reason=FailedCreate` shows the guard's message, and the install fails on a setup job or the wait in Verify (see online documentation) times out. The guard also denies node debug pods (`kubectl debug node/...`) in the `kamiwaza` namespace; start them in another namespace. `kubectl exec` or `kubectl attach` into `istio-init` is refused with `Forbidden` and the exec guard's message.

To remove the guard, delete the bindings first; that takes effect at once:

```
kubectl delete validatingadmissionpolicybinding kamiwaza-tenant-pod-guard kamiwaza-
tenant-exec-guard
kubectl delete validatingadmissionpolicy kamiwaza-tenant-pod-guard kamiwaza-tenant-
exec-guard
```

Kubernetes 1.28 and 1.29

These versions do not serve the v1 ValidatingAdmissionPolicy API, so applying the guard fails with `no matches for kind "ValidatingAdmissionPolicy" in version "admissionregistration.k8s.io/v1"`. Either upgrade the cluster to Kubernetes 1.30 or later and apply the guard, or enforce the same rules with the policy engine your cluster already runs, such as Kyverno or OPA Gatekeeper: Pod Security `baseline` in the Kamiwaza namespaces, with the `istio-init` exception exactly as the manifest above defines it, and a rule that refuses exec and attach into `istio-init`.

On OpenShift, the `restricted-v2` SecurityContextConstraints already deny privileged and `hostPath` pods to a namespace admin, so the guard is optional there.

Istio and the ingress gateway

Kamiwaza routes all traffic through an Istio mesh that the platform owns. Install and operate Istio the way your organization does; Kamiwaza does not install or configure it. The mesh must provide:

- **istiod, in sidecar mode.** Ambient mode is not supported.
- **An ingress gateway** exposed to your users on port 443, for example through a `LoadBalancer` Service or your existing load balancer.
- **An Istio `Gateway` resource** (`networking.istio.io`) that selects that gateway, serves the Kamiwaza domain on HTTPS port 443 with the edge certificate (see online documentation), and serves or redirects port 80. Its `hosts` must include the Kamiwaza domain: Istio programs a route only where the Gateway's hosts match, so a mismatch leaves the platform unreachable while the install reports success.

Kamiwaza attaches its routes to this Gateway from its own namespace. It writes nothing into the gateway's namespace.

A cluster without Istio. If the cluster has no mesh yet, for example a fresh k0s node with no load balancer, this example sets up one that meets the requirements above. It installs Istio 1.30.3 with Istio's Helm charts, and exposes the gateway on ports 80 and 443 of a node address through the Service's `externalIPs` instead of a `LoadBalancer`. Replace `<node-ip>` with the node address your users reach:

```
helm repo add istio https://blob.istio.io/istio-release/charts
helm repo update
helm upgrade --install istio-base istio/base --version 1.30.3 \
  --namespace istio-system --create-namespace --wait
helm upgrade --install istiod istio/istiod --version 1.30.3 \
  --namespace istio-system --wait \
  --set global.hub=docker.io/istio
helm upgrade --install istio-ingressgateway istio/gateway --version 1.30.3 \
  --namespace istio-system --wait \
  --set service.type=ClusterIP --set 'service.externalIPs={<node-ip>}'
```

`global.hub` sets the registry for istiod and for every proxy it injects, the ingress gateway's included. Keep it: the 1.30 chart's default registry, `registry.istio.io`, does not serve anonymous pulls. If you mirror Istio's images, set `global.hub` to your mirror, add its proxy repository to `istioInitImagePrefixes` in the pod guard (see **Mirrored proxy image**), and apply the guard again.

Kubernetes deprecates `externalIPs` as of 1.36, so expect the API server to warn when the gateway Service is created. The kube-proxy that k0s runs by default still routes it. If the cluster has a load balancer, leave out both `--set` options: the chart then creates a `LoadBalancer` Service.

Then create the Gateway. Save this as `kamiwaza-gateway.yaml`, with your domain in both `hosts` lists, and apply it with `kubectl apply -f kamiwaza-gateway.yaml`:

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: kamiwaza-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - kamiwaza.example.com
      tls:
        httpsRedirect: true
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - kamiwaza.example.com
      tls:
        mode: SIMPLE
        credentialName: kamiwaza-edge-tls
```

`kamiwaza-edge-tls` is the edge certificate (see online documentation) Secret, which you create in `istio-system`. The release name `istio-ingressgateway` and this Gateway match the gateway defaults in the **Record** table below: name, namespace, selector, service account, and Service. In DNS (see online documentation), point your domain at `<node-ip>`.

Check:

```
kubectl -n istio-system get deploy istiod
kubectl get crd virtualservices.networking.istio.io
authorizationpolicies.security.istio.io
kubectl auth can-i create virtualservices.networking.istio.io -n kamiwaza \
  --as=system:serviceaccount:kamiwaza:kamiwaza-installer   # yes
kubectl get gateways.networking.istio.io -A
kubectl -n <gateway-namespace> get gateways.networking.istio.io <gateway-name> -o yaml
# Expect a server on port 443, protocol HTTPS, whose hosts include your domain,
# and a tls.credentialName naming your edge certificate Secret
```

Record the gateway's coordinates. The install uses the defaults in the right column unless you record something different.

Fact	How to find it	Default
Gateway name and namespace	`kubectl get gateways.networking.istio.io -A`	`kamiwaza-gateway` in `istio-system`
Mesh namespace (where istiod runs)	`kubectl get deploy -A | grep istiod`	`istio-system`
Gateway pod selector	`spec.selector` of the Gateway resource. Record every label as-is.	`istio: ingressgateway`
Gateway workload's service account	`kubectl -n <ns> get pod -l <selector> -o jsonpath='{.items[0].spec.serviceAccountName}'`. A gateway installed with `istioctl` uses `istio-ingressgateway-service-account`.	`istio-ingressgateway`
Gateway Service's in-cluster hostname	`<service-name>.<namespace>.svc.cluster.local`	`istio-ingressgateway.istio-system.svc.cluster.local`
Sidecar style	See below	Native sidecars

Sidecar style. Kamiwaza expects Istio to inject `istio-proxy` as a Kubernetes native sidecar (an init container with `restartPolicy: Always`). If your mesh injects it as a regular container, record that: the install sets a value for it. To tell which, start a short-lived pod in the `kamiwaza` namespace and look for `istio-proxy`:

```
kubectl -n kamiwaza run kamiwaza-sidecar-check --image=busybox:1.37 --restart=Never --
sleep 300
kubectl -n kamiwaza wait --for=condition=Ready pod/kamiwaza-sidecar-check --
timeout=120s
kubectl -n kamiwaza get pod kamiwaza-sidecar-check \
  -o jsonpath='{range .spec.initContainers[*]}init {.name} {.restartPolicy}{"\n"}{end}
{range .spec.containers[*]}container {.name}{"\n"}{end}'
# "init istio-proxy Always": native sidecars.
# "container istio-proxy": classic injection.
# No istio-proxy line at all: the pod was not injected; check the namespace label.
kubectl -n kamiwaza delete pod kamiwaza-sidecar-check
```

A Pod Security `restricted` warning when this pod starts is expected; the namespace only warns and audits.

Edge certificate

The gateway serves the Kamiwaza domain with a TLS certificate that you provide and rotate. Create it as a `kubernetes.io/tls` Secret in the gateway's namespace and name it in the Gateway's `tls.credentialName`. The certificate must cover the Kamiwaza domain in its subject alternative names.

If the certificate is **not** issued by a publicly trusted CA (a private CA, or self-signed), Kamiwaza workloads need its CA to verify the gateway. Publish the CA certificate, or for a self-signed certificate the certificate itself, in the Kamiwaza namespace:

```
kubectl -n kamiwaza create configmap kamiwaza-edge-trust \
  --from-file=ca-certificates.crt=./edge-ca.crt
```

This ConfigMap holds public certificates only. Never put a private key in it. Kamiwaza reads it by name when it exists; no install value refers to it.

Check:

```
kubectl -n <gateway-namespace> get secret <edge-secret> -o jsonpath='{.data.tls\.crt}'
\
  | base64 -d | openssl x509 -noout -subject -issuer -ext subjectAltName -enddate
# Expect your domain under subjectAltName and an end date in the future
```

Record: whether the certificate is self-signed (subject and issuer are the same). The install sets a value for a self-signed certificate.

DNS

The Kamiwaza domain must resolve to the ingress gateway in two places:

- **For users:** your DNS points the domain at the gateway's external address.
- **Inside the cluster:** Kamiwaza services and apps call the platform at its public domain. Pods must resolve the domain to an address that reaches the gateway. If your cluster DNS forwards to a resolver that already answers for the domain, and pods can reach that address, nothing more is needed. Otherwise, add a cluster DNS entry that maps the domain to the gateway Service's in-cluster hostname.

With CoreDNS, add a `rewrite` line to the Corefile, above the `kubernetes` plugin:

```
rewrite name exact kamiwaza.example.com istio-ingressgateway.istio-
system.svc.cluster.local
```

Some distributions, k0s among them, manage the CoreDNS configuration and can put it back on restart or upgrade. On those, make the change through the distribution's own configuration. On k0s, add a patch under `spec.network.coreDNS.patches` in the k0s configuration that replaces the `coredns` ConfigMap's

`Corefile` with your edited copy (see k0s component patches (https://docs.k0sproject.io/stable/configuration/#component-patches)), then restart k0s so it reads the configuration again. Make the same change on every controller node.

First find the node's configuration file and the datastore k0s runs on. A restart with a configuration that names a different datastore brings the cluster up empty:

```
systemctl cat k0scontroller | grep ExecStart
# --config=<file> names the configuration file. Without --config, k0s reads
# /etc/k0s/k0s.yaml if it exists, and otherwise runs on built-in defaults
sudo grep -E '^ +type: (etcd|kine)$' /run/k0s/k0s.yaml
# The datastore the running k0s uses: etcd or kine
```

- **If the configuration file exists,** add the patch to it and leave `spec.storage` as it is.
- **If it does not,** create `/etc/k0s/k0s.yaml` with the patch and a `spec.storage.type` equal to the running datastore; k0s fills in defaults for everything else. Do not generate the file with `k0s config create`. Its output sets `type: etcd`, while a node started with `--single` runs on kine, and restarting that node with `type: etcd` brings it up on a new, empty etcd datastore.

Start from the current Corefile (`kubectl -n kube-system get configmap coredns -o jsonpath='{.data.Corefile}'`), add your line, and put the whole file in the patch. This is a complete file for a node that has none; to an existing file, add only `spec.network.coreDNS`:

```yaml
# /etc/k0s/k0s.yaml
apiVersion: k0s.k0sproject.io/v1beta1
kind: ClusterConfig
spec:
  storage:
    type: kine    # the datastore the check above printed
  network:
    coreDNS:
      patches:
        - target:
            kind: ConfigMap
            name: coredns
            namespace: kube-system
          patch:
            type: StrategicMergePatch
            content: |
              data:
                Corefile: |
                  .:53 {
                      # your complete edited Corefile
                  }
```

`type: kine` on its own selects k0s's default SOLite database under `/var/lib/k0s`. which is what `--single` uses. If `ExecStart` also sets `--data-dir`, copy `kine.dataSource` from `/run/k0s/k0s.yaml` into the file too.

Validate the file, take a backup, and restart k0s. `k0s backup` copies the configuration file, so it fails on a node that has none until you create it. The archive holds the datastore and the cluster's private keys: keep it off the node, somewhere safe. If the restart goes wrong, restore it as described in k0s backup and restore (https://docs.k0sproject.io/stable/backup/).

```
sudo "$(command -v k0s)" config validate --config /etc/k0s/k0s.yaml   # or the file --
config names
sudo "$(command -v k0s)" backup --save-path "$HOME"
sudo systemctl restart k0scontroller
```

After any cluster DNS change, allow a minute or two for CoreDNS to reload before running the check.

Check from your workstation, and from a pod:

```
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/
# Any HTTP status proves DNS and TLS reach the gateway. Before Kamiwaza is
# installed, expect 404.

kubectl -n default run kamiwaza-dns-check --restart=Never \
  --image=curlimages/curl --annotations=sidecar.istio.io/inject=false \
  --overrides='{"spec":{"securityContext":
{"runAsNonRoot":true,"runAsUser":100,"seccompProfile":
{"type":"RuntimeDefault"}},"containers":[{"name":"kamiwaza-dns-
check","image":"curlimages/curl","args":["-sk","-o","/dev/null","-w","%
{http_code}\\n","https://kamiwaza.example.com/"],"securityContext":
{"allowPrivilegeEscalation":false,"capabilities":{"drop":["ALL"]}}}]}}'
kubectl -n default wait --for=jsonpath='{.status.phase}'=Succeeded \
  pod/kamiwaza-dns-check --timeout=120s
kubectl -n default logs kamiwaza-dns-check
# Expect the same status from inside the cluster
kubectl -n default delete pod kamiwaza-dns-check
```

Record: the domain.

Block storage

Kamiwaza's databases and caches use `ReadWriteOnce` persistent volumes. The cluster must have a StorageClass that provisions them. Record its name; the install names it explicitly.

Size the storage provider for Kamiwaza's volume requests plus your provider's replication and free-space reserves. A StorageClass existing does not prove its volumes can be provisioned; the write test below does. Node-local storage, such as `local-path`, works on a single node but does not replicate data.

Any `reclaimPolicy` works, including `Delete`, the `local-path` default. The policy decides only what happens to a volume once its claim is deleted, as at uninstall (see online documentation).

Kamiwaza requests these volumes by default, about 624 GiB in total. Apps you deploy later add their own.

Volume	Default request	Holds	Change it with
`kamiwaza-registry-data`	500 GiB	Model and inference-engine cache	`registry.persistence.size`
`seaweedfs-data`	100 GiB	Built-in object store	`seaweedfs.persistence.size`
`core-postgres-data`	10 GiB	Platform database	—
`keycloak-postgres-data`	10 GiB	Identity database	—
`core-spicedb-postgres-data`	1 GiB	Authorization database	—
`core-etcd-data-core-etcd-0` to `-2`	1 GiB each	Configuration store	—

The model registry cache holds inference-engine images and the model files your users deploy. If you serve only external models (for example, through a cloud provider), or your storage is small, set `registry.persistence.size` lower, for example `100Gi`, in your install values before the first install. An existing volume keeps its size on upgrade.

Check the classes, then run a write test in the Kamiwaza namespace: write a marker, replace the pod, and read the marker back.

```
kubectl get storageclass
# Expect one class marked (default), or note the class you will name at install

cat > kamiwaza-storage-check.yaml <<'EOF'
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kamiwaza-storage-check
spec:
  accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 1Gi
  # storageClassName: <class>    # uncomment if you will not use the default
---
apiVersion: v1
kind: Pod
metadata:
  name: kamiwaza-storage-check
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    fsGroup: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: check
      image: busybox:1.37
      command: ["sh", "-c", "sleep 3600"]
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: ["ALL"]
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: kamiwaza-storage-check
EOF

kubectl -n kamiwaza apply -f kamiwaza-storage-check.yaml
kubectl -n kamiwaza wait --for=condition=Ready pod/kamiwaza-storage-check --timeout=5m
kubectl -n kamiwaza exec kamiwaza-storage-check -- sh -c 'echo ok > /data/marker'

# Replace the pod; the claim and its volume stay
kubectl -n kamiwaza delete pod kamiwaza-storage-check
kubectl -n kamiwaza apply -f kamiwaza-storage-check.yaml
kubectl -n kamiwaza wait --for=condition=Ready pod/kamiwaza-storage-check --timeout=5m
kubectl -n kamiwaza exec kamiwaza-storage-check -- cat /data/marker     # expect: ok

kubectl -n kamiwaza delete -f kamiwaza-storage-check.yaml
```

A class with `volumeBindingMode: WaitForFirstConsumer` binds the claim only once the pod is scheduled, so a `Pending` claim before the pod starts is normal.

Record: the StorageClass name.

Object storage

Workrooms, the Skills Library, and other features store objects. Choose one:

- **The built-in object store (default).** Kamiwaza runs SeaweedFS in its namespace, backed by a `ReadWriteOnce` volume from your block storage. Nothing to prepare.
- **Your own S3-compatible service.** See S3 workroom storage (see online documentation).

Kubernetes API addresses

Some Kamiwaza apps call the Kubernetes API from inside the namespace, and Kamiwaza allows that traffic with NetworkPolicy. Many CNIs, including the ones k0s and OpenShift use, evaluate NetworkPolicy **after** a Service address is translated to the API server's real address. So the policy needs both the `kubernetes` Service address **and** every API server endpoint address. With only the first, apps install cleanly and then fail at runtime.

Record both sets of addresses, as `/32` for IPv4 or `/128` for IPv6:

```
kubectl -n default get service kubernetes -o jsonpath='{.spec.clusterIPs[*]}{"\n"}'
kubectl -n default get endpointslices -l kubernetes.io/service-name=kubernetes \
  -o jsonpath='{.items[*].endpoints[*].addresses[*]}{"\n"}'
```

Copy the addresses exactly. A wrong address still installs cleanly, and the apps then fail at runtime. If the API server's addresses change (for example, a control-plane node is replaced), update them in the install values and upgrade.

If your CNI enforces NetworkPolicy **before** Service address translation, the endpoint addresses are not needed; the Service address alone works. Record it anyway.

Registry access

Kamiwaza's charts and images are published to a public registry and pull without credentials. Nodes must be able to pull from it; see Outbound network access (see online documentation).

If you mirror images into your own registry and it requires credentials, create a `kubernetes.io/dockerconfigjson` pull Secret in the Kamiwaza namespace. Kamiwaza uses only namespaced pull Secrets, never node-wide registry credentials. Build the Secret from a file, so the password never appears in a command line or your shell history:

```
read -rs REGISTRY_PASSWORD      # typed, not echoed
(umask 077; d=$(mktemp -d)
 printf '{"auths":{"%s":{"auth":"%s"}}}' '<registry-host>' \
   "$(printf '%s:%s' '<username>' "$REGISTRY_PASSWORD" | base64 | tr -d '\n')" >
"$d/config.json"
 kubectl -n kamiwaza create secret generic kamiwaza-pull \
   --type=kubernetes.io/dockerconfigjson --from-file=.dockerconfigjson="$d/config.json"
 rm -rf "$d")
unset REGISTRY_PASSWORD
```

`<registry-host>` is the registry's host name, with its port if it has one, as it appears in the image references.

If the registry is GitHub Container Registry (`ghcr.io`), the password is a GitHub token that carries the `read:packages` scope for the organization that owns the images. A token without it, such as one from `gh auth login` with its default scopes, is refused, and every pod that uses a private image goes to `ImagePullBackOff` with `403 Forbidden ... The token provided does not match expected scopes` while the install keeps waiting.

A build that has not been published to the public registry yet, such as a pre-release Kamiwaza provided by your Kamiwaza contact, pulls its images from a private registry and needs this Secret. Create it before you run the install. If your Kubernetes substrate was prepared with Kamiwaza's setup automation, its hand-off values name the Secret in `global.imagePullSecrets` (`kamiwaza-ghcr-pull`); create the Secret under that name instead of `kamiwaza-pull`.

Record: the pull Secret's name, `kamiwaza-pull` above. The install lists it in `global.imagePullSecrets`. That list does not reach the engine pods of tenant-mode inference (see online documentation): they carry only the staging credential, which defaults to the first entry in `global.imagePullSecrets`.

Node kernel limits

Every Istio sidecar watches files with inotify. Distribution defaults are too low for the number of pods Kamiwaza runs, and sidecars then crash with `too many open files`. Set these floors on **every node**, persistently:

```
cat <<'EOF' | sudo tee /etc/sysctl.d/99-kamiwaza-inotify.conf
fs.inotify.max_user_instances = 8192
fs.inotify.max_user_watches = 1048576
fs.inotify.max_queued_events = 16384
EOF
sudo sysctl --system
```

Check on each node:

```
sysctl fs.inotify.max_user_instances fs.inotify.max_user_watches
fs.inotify.max_queued_events
# Expect values at or above 8192, 1048576, and 16384
```

Managed platforms that do not give you node access usually expose these limits through a node configuration resource; use your platform's mechanism.

Outbound network access

Nodes and pods need outbound HTTPS (port 443) to:

- `ghcr.io` and `pkg-containers.githubusercontent.com`, for Kamiwaza charts and images;
- the model sources you use, for example `huggingface.co` and its content hosts, or your cloud model provider's endpoint.

The checks on this page use the public `busybox` and `curlimages/curl` images from Docker Hub. If your cluster cannot reach Docker Hub, mirror them to your registry.

For installs with no outbound access, see Install Without Internet Access (see online documentation).

GPU nodes

GPU inference is optional. To serve models on NVIDIA GPUs, prepare each GPU node as described in GPU Nodes (see online documentation): a driver that meets the engine images' floor, persistence mode, the NVIDIA container runtime and the `nvidia` RuntimeClass, the NVIDIA GPU Operator, and no `NoSchedule` or `NoExecute` taint on the node. Kamiwaza installs none of these.

Prepared nodes are not enough on their own: a default install cannot use GPUs. After the install, the cluster owner turns GPU serving on with Tenant-mode inference (see online documentation).

Check:

```
kubectl get nodes -o custom-
columns='NODE:.metadata.name,GPU:.status.allocatable.nvidia\.com/gpu'
# Expect a count on each GPU node
kubectl get runtimeclass nvidia
# Expect a RuntimeClass named nvidia, with handler nvidia
```

Record: which nodes have GPUs, and the GPU model on each. The cluster owner needs them to publish a signed profile.

Multi-node clusters

On a cluster with more than one node, also provide:

- **Storage that survives a node loss.** Use a StorageClass whose volumes are replicated or network-attached, such as a CSI driver for your SAN or cloud block storage, or a replicated provider like Longhorn. Node-local storage ties each database to one node: if that node is lost, so is the data.
- **Every GPU node prepared.** Prepare each GPU node as in GPU nodes (see online documentation), including nodes added later.

- **A CNI that enforces NetworkPolicy.** Kamiwaza isolates apps and jobs from each other with NetworkPolicy. A CNI that accepts policies but does not enforce them leaves that isolation off without any error.
- **The inotify floor on every node**, including nodes added later.
- **The API endpoint addresses of every control-plane node** in the recorded Kubernetes API addresses (see online documentation).

Next step

When every check passes, hand the recorded facts and the installer kubeconfig to whoever runs the install, and continue to Install on Kubernetes with Istio (see online documentation).

Install on Kubernetes with Istio

This page installs Kamiwaza into one namespace of a Kubernetes cluster that runs an Istio service mesh. You run a single `helm upgrade --install` as a namespace-scoped identity. The chart creates only namespaced objects.

Before you start

You need	Notes
A prepared cluster	Every check in Platform Prerequisites (see online documentation) passes, and you have the facts it told you to record.
The installer kubeconfig	The namespace-scoped identity from Installer identity (see online documentation). Never install with a cluster-admin kubeconfig.
`kubectl`, and Helm 3.14 or later	Helm pulls the chart from an OCI registry. Helm 4 also works.
`jq` and `curl`	Used by the verification steps.
A license file	`license.lic`, issued by Kamiwaza. See License File (see online documentation).

No internet access? Do Install Without Internet Access (see online documentation) first. It copies the release into your registry and gives you values to merge into your values file in step 2. In step 3, it has you check the values file before you install, and use the chart from your registry.

Point `kubectl` and `helm` at the installer identity for every command on this page:

```
export KUBECONFIG=<path to the installer kubeconfig>
```

Kamiwaza installs into the `kamiwaza` namespace. The examples use `kamiwaza.example.com` as the domain; substitute your own.

Step 1: Add the license

Create a Secret from the license file. The key must be `license.lic`:

```
kubectl -n kamiwaza create secret generic kamiwaza-license \
  --from-file=license.lic=./license.lic
```

The chart refuses to render until a license is configured and the EULA is accepted. You do both in the values file in the next step.

Step 2: Write your values file

Your values file carries this cluster's facts and your acceptance of the EULA. Everything else is the chart's defaults. Keep the file somewhere safe and outside any shared repository; you will need it again for upgrades.

Required values

Value	Set it to
`global.eula.accepted`	`true`, after reading the EULA (https://www.kamiwaza.ai/eula-license). The render fails without it.
`core.license.existingSecret`	`kamiwaza-license`, the Secret from step 1. The render fails without it.
`global.domain`	The Kamiwaza domain the gateway serves.
`global.namespaces.platform`, `.extensions`, `.sandboxes`, `.observability`, `.ca`, `.registry`, `.system`	`kamiwaza`, for each one. Everything runs in that one namespace.
`global.namespaces.mesh`	The namespace istiod runs in, usually `istio-system`.
`global.storage.stateful.standard`, `.postgres`, `.etcd`, `.registry`, `.extensions`	The StorageClass you recorded, for each one.
`core.extensionRuntime.kubernetesApi.egressCidrs`	Every Kubernetes API Service address **and** every endpoint address you recorded, each as `/32` (IPv4) or `/128` (IPv6). Copy them exactly: a wrong address installs cleanly and then fails at runtime. The render fails without this value.
`global.ingress.gateway.serviceAccountName`	The gateway workload's service account you recorded. A gateway installed with Istio's Helm chart as `istio-ingressgateway` uses `istio-ingressgateway`: one installed with `istioctl` uses `istio-ingressgateway-service-account`. A wrong value makes Kamiwaza reject the gateway's traffic.

Values that depend on your cluster

Value	Set it when
`global.ingress.gateway.name`, `.namespace`	The Gateway resource is not `kamiwaza-gateway` in `istio-system`.
`global.ingress.gateway.selector`	The Gateway's `spec.selector` is anyt but exactly `istio: ingressgateway`, t default. List all of its labels.
`global.ingress.gateway.serviceHostname`	The gateway Service is not `istio-ingressgateway.istio-system.svc.cluster.local`.
`global.istio.classicInjection`	Your mesh injects `istio-proxy` as a re container, not a native sidecar. Set `true` such a mesh, leaving it `false` makes th install hang on its setup jobs.
`core.scheduler.extraEnv`	The edge certificate is self-signed. Add th `KAMIWAZA_ALLOW_INSECURE_INTERNAL_` entry exactly as shown in the example be Omit it for a CA-issued certificate.
`global.imagePullSecrets`	Images come from a registry that needs credentials. List the pull Secret you create example `[{name: kamiwaza-pull}]`.
`core.inferenceResources.enabled`, `.bundleConfigMap`, `.ownerPublicKey`, `.clusterBindingId`	You serve models on GPUs. Set them aft cluster owner publishes a signed profile, Tenant-mode inference (see online documentation). Without them, a GPU deployment is refused. Prepare the node see GPU nodes (see online documentation)
`core.extensionRuntime.kubernetesApi.requireEgressCidrs`	Your CNI enforces NetworkPolicy before address translation and you choose not t the API addresses. Set `false`. Most clu including k0s and OpenShift, need the addresses.

Optional values

Value	Notes
`registry.persistence.size`, `seaweedfs.persistence.size`	Volume sizes for the model registry cache (default `500Gi`) and the built-in object store (default `100Gi`). Lower them when storage is small; see Block storage (see online documentation). Set them before the first install; they can't be reduced later.
`core.userDefaults.adminPassword`	The password for the `admin` user. Leave it unset and Kamiwaza generates one, which you read back in step 4 (see online documentation). Setting it here stores it in the Helm release. It applies at first install only; changing it later does not rotate the password.

Example

```yaml
global:
  domain: kamiwaza.example.com
  eula:
    accepted: true
  namespaces:
    platform: kamiwaza
    extensions: kamiwaza
    sandboxes: kamiwaza
    observability: kamiwaza
    ca: kamiwaza
    registry: kamiwaza
    system: kamiwaza
    mesh: istio-system
  storage:
    stateful:
      standard: standard
      postgres: standard
      etcd: standard
      registry: standard
      extensions: standard
  ingress:
    gateway:
      serviceAccountName: istio-ingressgateway   # the service account you recorded

core:
  license:
    existingSecret: kamiwaza-license
  extensionRuntime:
    kubernetesApi:
      egressCidrs:
        - 10.96.0.1/32        # kubectl -n default get service kubernetes
        - 192.168.10.15/32    # the kubernetes EndpointSlice addresses

  # Only for a self-signed edge certificate:
  # scheduler:
  #   extraEnv:
  #     - name: KAMIWAZA_ALLOW_INSECURE_INTERNAL_RETRY
  #       value: "true"
```

Save it as `kamiwaza-values.yaml`.

Step 3: Install

Record the start time first. The verification step uses it to ignore events from before this install.

```
INSTALL_START=$(date -u +%Y-%m-%dT%H:%M:%SZ)

helm upgrade --install kamiwaza \
  oci://ghcr.io/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza \
  --version <version> \
  --namespace kamiwaza \
  --values kamiwaza-values.yaml \
  --timeout 70m
```

`<version>` is the Kamiwaza version you are installing, for example `1.3.4`.

- Do not add `--create-namespace`. The namespace belongs to the platform administrator.
- Helm may print Pod Security `restricted` warnings, `EnvoyFilter exposes internal implementation details` warnings, and `WARNING: core cannot see GPUs on this install.` until tenant-mode inference is on. All are expected.
- A typical install takes 5 to 15 minutes, most of it pulling images.

Install and upgrade with Helm, not rendered manifests

Always install and upgrade with `helm upgrade --install` against the cluster. Rendering the chart with `helm template` or `--dry-run` and applying the output, for example through Argo CD or Flux, is not supported. On first install the chart generates the platform's encryption keys, signing keys, and database passwords, and on each upgrade it reads them back from the cluster to keep them. A rendered chart cannot read the cluster, so every render generates new ones. Applying them makes existing encrypted data and access tokens unreadable.

To upgrade to a later release in the same line, rerun the command in this step with the new `--version` and the same values file.

Upgrading with Helm 4: changes made with kubectl

Helm 4 installs with server-side apply, and its upgrades keep the method of the release's last revision. Under server-side apply, Helm cannot change a field that someone else last changed, for example an image set with `kubectl set image` or a field changed with `kubectl edit` or `kubectl patch`. The upgrade then stops part-way and records a failed revision:

```
Error: UPGRADE FAILED: conflict occurred while applying object kamiwaza/core-scheduler
apps/v1, Kind=Deployment:
Apply failed with 1 conflict: conflict with "kubectl-set" using apps/v1:
.spec.template.spec.containers[name="core"].image
```

Check before upgrading, with the installer kubeconfig. First see how the release was last applied:

```
helm get metadata kamiwaza -n kamiwaza | grep APPLY_METHOD
```

If it says `client-side apply`, or prints nothing (Helm 3), the upgrade uses client-side apply and cannot hit this; skip the next command. If it says `server-side apply`:

```
helm get manifest kamiwaza -n kamiwaza \
  | kubectl -n kamiwaza apply --server-side --dry-run=server \
      --field-manager=helm --validate=false -f - 2>&1 >/dev/null \
  | grep -v '^Warning:'
# Expect no output. Nothing is changed in the cluster.
```

If it prints `conflict with "kubectl-..."`, or an upgrade has already failed with that error, choose one:

- Add `--server-side=false` to the upgrade command. Helm uses client-side apply and sets the chart's values over the changed fields. Later upgrades keep client-side apply, so you do not need the flag again. After a failed upgrade, rerun the same command with the flag; no rollback is needed.
- Or add `--force-conflicts`. Helm takes over those fields and keeps server-side apply. Do not combine it with `--force` or `--force-replace`, and do not use it when the conflicting manager is not `kubectl...`: a controller writes its field back, and the next upgrade fails again. Use `--server-side=false` instead.

Either way the change made with kubectl is replaced by the chart's value. If you need to keep it, put it in your values file first.

Step 4: Verify

Check what the platform does, not only its status fields.

Wait for the platform to start. Helm can return before every pod is ready and every setup job has finished:

```
kubectl -n kamiwaza wait deployment -l app.kubernetes.io/managed-by=Helm \
  --for=condition=Available --timeout=30m
kubectl -n kamiwaza wait job -l app.kubernetes.io/managed-by=Helm \
  --for=condition=Complete --timeout=70m
```

Pods are running and ready:

```
kubectl -n kamiwaza get pods --no-headers \
  | awk '$3 != "Completed" { split($2, r, "/"); if ($3 != "Running" || r[1] != r[2])
print }'
# Expect no output
```

An `Error` pod whose Job later completed is a retried attempt, not a failure. Confirm with `kubectl -n kamiwaza get jobs`: every Job should show all completions.

No image pulls failed during this install. Older events in the namespace are ignored. Run this in the same shell as step 3, or set `INSTALL_START` to the time you started the install, in the same format:

```
kubectl -n kamiwaza get events -o json | jq --arg start "$INSTALL_START" '
  [.items[]
   | select((.lastTimestamp // .eventTime // .metadata.creationTimestamp) >= $start)
   | select((.message // "") | test("ErrImagePull|ImagePullBackOff|Failed to pull
image|Back-off pulling image"))
  ] | length'
# Expect 0
```

The platform answers at its domain:

```
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/       # expect
200
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/api/   # expect
401
```

Log in. If you did not set an admin password, read the generated one:

```
kubectl -n kamiwaza get secret kamiwaza-user-admin \
  -o jsonpath='{.data.password}' | base64 -d; echo
```

Open `https://kamiwaza.example.com/` and log in as `admin`. With a certificate your browser does not trust, accept the warning first.

`admin` is the Kamiwaza application administrator. It is not the installer identity and not the Keycloak management account.

The platform is installed. Deploying models and apps is covered in the Quickstart (see online documentation).

Troubleshooting

The install stops before creating anything. The chart checks its values before rendering. A missing EULA acceptance or license prints `INSTALLATION HALTED`; a missing `egressCidrs` value names the value it needs. Nothing was installed. Fix the values file and rerun step 3.

`no Kamiwaza license is configured`, with the license set. The license value belongs under `core:`. Check that it is `core.license.existingSecret`, not `license.existingSecret`. To see values Helm received but no chart reads, run `helm get values kamiwaza -n kamiwaza`.

The install hangs for about 15 minutes, then fails on a post-install job. This is almost always a cascade from `core-scheduler` not starting. Check its logs first: `kubectl -n kamiwaza logs deploy/core-scheduler`. A license problem shows here as `License check failed`; see License File (see online documentation). While the job runs, every `helm upgrade` fails with `another operation (install/upgrade/rollback) is in progress`.

Setup jobs stay at `1/2` ready and the install never finishes. The mesh uses classic sidecar injection. Set `global.istio.classicInjection: true` and rerun step 3.

Pods are `Pending` on their volumes. A StorageClass named in the values does not exist or cannot provision. Compare `kubectl get storageclass` with `global.storage.stateful.*`.

Images fail to pull with `unauthorized`. The pull Secret named in `global.imagePullSecrets` does not exist in the namespace, or has the wrong credentials. Compare it with `kubectl -n kamiwaza get secret`. If the pull fails with `403 Forbidden ... does not match expected scopes`, the token lacks `read:packages`; see Registry access (see online documentation). Recreate the Secret with a token that has it. The pods retry the pull with back-off, so an install that has not yet timed out continues without a rerun; if the install already timed out, run the same install command again after recreating the Secret.

The platform returns 404 at its domain. The Gateway's `hosts` do not include `global.domain`, or `global.ingress.gateway.*` does not name your Gateway. See Istio and the ingress gateway (see online documentation).

Apps install but cannot reach the Kubernetes API. `egressCidrs` is missing the endpoint addresses. Add every address from Kubernetes API addresses (see online documentation) and rerun step 3.

An app returns "Page Not Found", or the platform's own page. Wait about 40 seconds after its pods are ready. If the app stays deployed with no route, delete it and deploy it again.

A GPU model deployment fails with HTTP 422 and `Core cannot see a GPU on this install`. Tenant-mode inference is off, which is the default. Prepare the GPU nodes, then turn it on; see Tenant-mode inference (see online documentation).

Next steps

- Quickstart (see online documentation): start using Kamiwaza.
- License File (see online documentation): check and rotate the license.
- Uninstall (see online documentation).

Uninstall from Kubernetes with Istio

This page removes a Kamiwaza installation from the `kamiwaza` namespace. It does not change the platform around it: the namespace, the service mesh and gateway, storage classes, DNS, and certificates belong to the platform administrator and stay in place.

> **This is destructive.** Uninstalling deletes Kamiwaza's databases, stored objects, and model files along with the platform. Back up anything you need before you begin.

Step 1: Back up

Export or copy any data you need to keep, such as documents in workrooms, configuration, and anything in the platform databases. Step 3 deletes the platform databases and object store, and they cannot be recovered after it.

Step 2: Remove apps and models

Remove what you deployed through Kamiwaza while the platform is still running:

1. Log in as `admin`.
2. In the App Garden, stop every deployed app. In the Tool Shed, stop every tool server.
3. On the Models page, stop every model deployment.
4. Confirm that all three lists are empty.

Step 3: Uninstall the release

Run as the installer identity. This deletes the volume claims for the platform databases and the object store; step 4 describes what happens to the disks behind them:

```
helm uninstall kamiwaza --namespace kamiwaza --wait --timeout 15m
```

Step 4: Delete the remaining volumes

Two kinds of volume outlive the release: the model registry cache, which the chart keeps on purpose, and the configuration store's volumes, which Kubernetes keeps for any StatefulSet. Delete them:

```
kubectl -n kamiwaza get persistentvolumeclaims
kubectl -n kamiwaza delete persistentvolumeclaims --all
```

Whether the underlying disks are also deleted depends on the StorageClass's `reclaimPolicy`. With `Retain`, the platform administrator deletes the released PersistentVolumes.

Step 5: Verify

Run as the platform administrator, since the installer identity may not be able to list every kind:

```
kubectl api-resources --verbs=list --namespaced -o name \
  | grep -v -E '^events(\.events\.k8s\.io)?$' \
  | xargs -n 1 kubectl -n kamiwaza get --ignore-not-found -o name 2>/dev/null
```

Expect only the platform prerequisites and what Kubernetes and Istio create in every namespace:

- the installer's ServiceAccount, Roles, and RoleBindings;
- the `kamiwaza-license` Secret, and your pull Secret if you use one;
- the `kamiwaza-edge-trust` ConfigMap, if you use one;
- the `kube-root-ca.crt`, `istio-ca-root-cert`, and (on Istio versions that publish one) `istio-ca-crl` ConfigMaps, and the `default` ServiceAccount.

If anything else remains, contact Kamiwaza support with the list before reinstalling into the same namespace.

Removing the platform prerequisites

If Kamiwaza will not be installed again, the platform administrator can remove what was prepared for it, following their own procedures for the mesh, storage, DNS, and certificates:

- the `kamiwaza` namespace, which removes the installer identity, license Secret, and everything else in it:

  ```
  kubectl delete namespace kamiwaza
  ```

- the Istio `Gateway` and edge certificate Secret, if they serve only Kamiwaza;
- the DNS records for the Kamiwaza domain, including any in-cluster DNS entry;
- the inotify settings in `/etc/sysctl.d/99-kamiwaza-inotify.conf` on each node, if nothing else relies on them.

To install again after deleting the namespace, start from Platform Prerequisites (see online documentation).

Platform Prerequisites for OpenShift with GreyMatter

Kamiwaza installs into a single namespace on an OpenShift cluster that you already run, with GreyMatter as its service mesh. Before the install, a **platform administrator** (someone with cluster-admin rights, and credentials for your GreyMatter platform) prepares the cluster so it provides everything on this page. The Kamiwaza chart does not create or repair any of it: it creates no cluster-scoped objects, installs no operators, and does not configure OpenShift or GreyMatter. If a prerequisite is missing, the install fails or the platform comes up unable to serve traffic.

Work through each section and run its check. Each section also says what to **record**; the install (see online documentation) needs those facts.

Hardware sizing and supported operating systems are covered separately, in System Requirements (see online documentation).

Summary

Prerequisite	What the cluster must provide
Namespace (see online documentation)	A pre-created project
Installer identity (see online documentation)	A namespace-scoped identity to run Helm as. Never cluster-admin
Security context constraints (see online documentation)	OpenShift's default `restricted-v2` for Kamiwaza's service accounts, and no custom SCC
GreyMatter (see online documentation)	The namespace enrolled in GreyMatter, and a Git repository for Kamiwaza's GreyMatter configuration
Edge certificate (see online documentation)	A TLS certificate for the domain, issued by your GreyMatter mesh CA and valid for server and client authentication
DNS (see online documentation)	The domain resolves to the OpenShift router, from clients and from inside the cluster
Block storage (see online documentation)	An RWO StorageClass that provisions durable volumes
Object storage (see online documentation)	A decision: the built-in object store, or your own S3-compatible service
Kubernetes API addresses (see online documentation)	The API server's Service and endpoint addresses, recorded for network policy
Registry access (see online documentation)	Nodes can pull Kamiwaza images and the GreyMatter proxy image
Node kernel limits (see online documentation)	`fs.inotify` limits at or above the floor on every node
Outbound network access (see online documentation)	HTTPS to the image registry, your Git server, and any model sources you use

GPU inference and multi-node clusters add requirements; see GPU nodes (see online documentation) and Multi-node clusters (see online documentation).

Kamiwaza installs into a namespace named `kamiwaza`; use that name. The examples use `kamiwaza.example.com` as the domain; substitute your own. Run the checks with the administrator's kubeconfig unless a check says otherwise. `oc` works everywhere this page uses `kubectl`.

Namespace

Create the `kamiwaza` namespace. The installer does not create it, and the install never uses `--create-namespace`.

Do not label it for Istio injection: GreyMatter injects its own sidecars once the namespace is enrolled (see GreyMatter (see online documentation)). OpenShift applies pod security through security context constraints, covered below (see online documentation).

```
oc new-project kamiwaza
oc label namespace kamiwaza app.kubernetes.io/part-of=kamiwaza
```

`oc new-project` also switches your kubeconfig's current namespace to `kamiwaza`. Switch back with `oc project <previous namespace>` if you need to.

Check:

```
oc get namespace kamiwaza --show-labels
# Expect app.kubernetes.io/part-of=kamiwaza, and no istio-injection label
```

Installer identity

Helm runs as a namespace-scoped identity, never as cluster-admin. It needs:

- the built-in `admin` ClusterRole, bound **in the namespace only** with a RoleBinding;
- rights to manage `endpoints`, patch `pods/status`, and read GreyMatter's `sidecarresources`. The chart grants these to its own service accounts, and Kubernetes only lets an identity grant permissions it holds itself. Without the `sidecarresources` read, the install stops with `attempting to grant RBAC permissions not currently held`.

This manifest creates a service account with those rights. You can bind the same Roles to a user or group from your identity provider instead.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kamiwaza-installer
  namespace: kamiwaza
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kamiwaza-installer-admin
  namespace: kamiwaza
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
  - kind: ServiceAccount
    name: kamiwaza-installer
    namespace: kamiwaza
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kamiwaza-installer-delegation
  namespace: kamiwaza
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["create", "delete", "patch", "update"]
  - apiGroups: [""]
    resources: ["pods/status"]
    verbs: ["patch"]
  - apiGroups: ["tenant.greymatter.io"]
    resources: ["sidecarresources"]
    verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kamiwaza-installer-delegation
  namespace: kamiwaza
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: kamiwaza-installer-delegation
subjects:
  - kind: ServiceAccount
    name: kamiwaza-installer
    namespace: kamiwaza
```

Save it as `kamiwaza-installer.yaml` and apply it as the administrator:

```
kubectl apply -f kamiwaza-installer.yaml
```

Give the installer a kubeconfig for this identity, following your cluster's normal practice for issuing credentials. For example, with a service account token, run as the administrator:

```
(
  set -e
  SERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
  kubectl config view --minify --raw \
    -o jsonpath='{.clusters[0].cluster.certificate-authority-data}' | base64 -d >
cluster-ca.crt
  TOKEN=$(kubectl -n kamiwaza create token kamiwaza-installer --duration=24h)

  export KUBECONFIG=kamiwaza-installer.kubeconfig
  kubectl config set-cluster kamiwaza --server="$SERVER" \
    --certificate-authority=cluster-ca.crt --embed-certs
  kubectl config set-credentials kamiwaza-installer --token="$TOKEN"
  kubectl config set-context kamiwaza --cluster=kamiwaza --user=kamiwaza-installer --
namespace=kamiwaza
  kubectl config use-context kamiwaza
)
```

The parentheses keep these commands from changing your own shell's `KUBECONFIG`. If your kubeconfig names a `certificate-authority` file instead of embedding `certificate-authority-data`, copy that file to `cluster-ca.crt` instead. The token expires, and the API server may cap the duration you ask for. Issue a new one before each later upgrade or uninstall.

Check that the identity can create namespaced resources and cannot create cluster-scoped ones:

```
SA=system:serviceaccount:kamiwaza:kamiwaza-installer
kubectl auth can-i create deployments -n kamiwaza --as=$SA            # yes
kubectl auth can-i patch pods/status -n kamiwaza --as=$SA             # yes
kubectl auth can-i get sidecarresources.tenant.greymatter.io -n kamiwaza --as=$SA   #
yes
kubectl auth can-i create clusterroles --as=$SA                       # no
kubectl auth can-i create customresourcedefinitions --as=$SA          # no
kubectl auth can-i list nodes --as=$SA                                # no
kubectl auth can-i create securitycontextconstraints --as=$SA         # no
```

When your kubeconfig sets a namespace, the cluster-scoped checks also print `Warning: resource '...' is not namespace scoped` before their answer. The `yes` or `no` is the result. The security context constraint check below passes `-n` itself, so it always prints that warning.

Security context constraints

Kamiwaza runs under OpenShift's default `restricted-v2` security context constraint (SCC), which assigns each pod a user ID from the namespace's range. The chart creates no SCCs and needs none.

Don't create or bind a custom SCC for Kamiwaza, and don't grant `anyuid`, `nonroot`, or a similar SCC to its service accounts. Kamiwaza's pods rely on `restricted-v2` assigning their user ID; under an SCC that doesn't, they fail to start with errors such as `container has runAsNonRoot and image has non-numeric user` or `image will run as root`.

Check that no permissive SCC is granted to the whole namespace or to all service accounts. It works before the install creates Kamiwaza's service accounts:

```
for scc in anyuid nonroot nonroot-v2 privileged; do
  printf '%s: ' "$scc"
  oc auth can-i use "securitycontextconstraints/$scc" -n kamiwaza \
    --as=system:serviceaccount:kamiwaza:core-scheduler \
    --as-group=system:serviceaccounts --as-group=system:serviceaccounts:kamiwaza
done
# Expect "no" for each
```

A grant to a single service account shows up in the install's verification (see online documentation), which checks that every Kamiwaza pod runs under `restricted-v2`.

GreyMatter

Kamiwaza routes all traffic through a GreyMatter mesh that the platform owns. GreyMatter Core 2.4 must already be installed and healthy; Kamiwaza does not install or configure it. The steps below change GreyMatter configuration, so they need your GreyMatter credentials.

GreyMatter configures Kamiwaza's routes from configuration files (GSL) in a Git repository. Kamiwaza supplies those files: you publish them at install time, and Kamiwaza adds routes to them at runtime as apps and models are deployed. The browser-facing entry point is a GreyMatter edge in the Kamiwaza namespace, published through an OpenShift Route with TLS passthrough.

Enroll the namespace

In your GreyMatter Core configuration, add the namespace to `tenant_namespaces` in `config.cue`. If your configuration also sets `ftr_namespaces`, add it there too. Leave `ftr_namespaces` unset if it is: when it is absent, forced traffic redirection already covers every tenant namespace, and setting it limits redirection to the namespaces it lists.

```
config: {
  tenant_namespaces: [
    "kamiwaza",
  ]
}
```

The namespace must exist before GreyMatter receives this change. Apply it through your normal GreyMatter change process.

Check that GreyMatter has acted on the namespace. It starts its tenant controllers there and copies its image pull Secret in:

```
oc -n kamiwaza get deployments
# Expect the GreyMatter tenant controllers, for example greymatter-tcm
oc -n kamiwaza get secret greymatter-image-pull -o jsonpath='{.type}{"\n"}'
# Expect kubernetes.io/dockerconfigjson
```

The tenant controllers stay in `ContainerCreating` until the edge certificate (see online documentation) Secret exists: they mount it. Check that they are ready after that section.

Record:

- the GreyMatter image pull Secret's name, if it is not `greymatter-image-pull`;
- the GreyMatter proxy image your platform runs. Kamiwaza runs the same proxy beside its Ray workloads:

  ```
  oc get deployments -A \
    -o jsonpath='{range .items[*].spec.template.spec.containers[*]}{.image}{"\n"}
  {end}' \
    | grep -i greymatter-proxy | sort -u
  ```

Tenant configuration repository

Provide a Git repository for Kamiwaza's GreyMatter configuration, with a branch that GreyMatter Sync watches. Kamiwaza pushes route changes to that branch while it runs, so the branch must accept direct pushes, not only pull requests. Kamiwaza's files live under `_tenants/kamiwaza` in the repository.

Create the GreyMatter project there with the `greymatter` CLI, and push it. Use the CLI that matches your GreyMatter platform: the `.greymatter` file at the root of your GreyMatter Core configuration records the platform and CLI versions. GreyMatter distributes the CLI to its customers; if the exact CLI build isn't offered, use the newest patch release of the same minor version.

The CLI writes the project into the current directory, so create the project directory first:

```
git clone https://git.example.com/org/kamiwaza-gsl.git && cd kamiwaza-gsl
git switch -c tenant/kamiwaza
mkdir -p _tenants/kamiwaza && cd _tenants/kamiwaza
greymatter create project --security pki --openshift kamiwaza
cd ../..
git add _tenants/kamiwaza
git commit -m "Create the Kamiwaza GreyMatter project"
git push -u origin tenant/kamiwaza
```

The CLI writes `.greymatter` and `cue.mod/`, including the GSL packages that match your GreyMatter version, and a starter `greymatter/` directory. Commit them unchanged. Kamiwaza's own files are added to `greymatter/` at install time, replacing the starter files.

Then create the Secret that connects GreyMatter Sync and Kamiwaza to the repository. It holds the repository's location and a credential that can push to the branch. Read the credential without leaving it in your shell history:

```
printf 'Git username: '; read -r GIT_USER
printf 'Git token: '; read -rs GIT_TOKEN; echo
(umask 077; d=$(mktemp -d)
 printf '%s' "$GIT_TOKEN" > "$d/token"
 oc -n kamiwaza create secret generic greymatter-tenant-repo \
   --type=greymatter.io/repo \
   --from-literal=auth_type=https \
   --from-literal=url=https://git.example.com/org/kamiwaza-gsl.git \
   --from-literal=branch=tenant/kamiwaza \
   --from-literal=relative_path=_tenants/kamiwaza \
   --from-literal=http_username="$GIT_USER" \
   --from-file=http_password="$d/token" \
   --from-literal=tls_insecure_verify=false
 rm -rf "$d")
unset GIT_USER GIT_TOKEN
```

If your Git server uses a private CA, add `--from-file=tls_remote_ca=<ca-bundle>`. Keep `tls_insecure_verify=false`.

Check:

```
oc -n kamiwaza get secret greymatter-tenant-repo -o json \
  | jq -r '.type, (.data | keys | join(" "))'
# Expect greymatter.io/repo, then the keys
# auth_type branch http_password http_username relative_path tls_insecure_verify url
# (and tls_remote_ca, if you added it)
```

Record: the repository URL and branch. Whoever publishes Kamiwaza's files at install time needs push access to them.

Network policy

If the `kamiwaza` namespace denies traffic by default, allow:

- DNS to OpenShift DNS;
- the Kubernetes API, at the addresses you record below (see online documentation);
- HTTPS to your Git server, from the GreyMatter tenant controllers and from Kamiwaza's `core-scheduler` and `core-raycluster` pods;
- GreyMatter control-plane traffic to and from the namespace;
- the OpenShift router to the GreyMatter edge;
- HTTPS to the OpenShift router, for calls to the Kamiwaza domain from inside the cluster (see DNS (see online documentation));
- HTTPS to the hosts in Outbound network access (see online documentation);
- traffic within the namespace.

Scope each egress rule to the addresses it needs, not to every address.

Edge certificate

The Kamiwaza domain is served with a TLS certificate that you provide and rotate. The certificate must cover the Kamiwaza domain in its subject alternative names.

The GreyMatter edge serves it; the OpenShift router passes the connection through without decrypting it. The same certificate is also the mesh identity in the Kamiwaza namespace: GreyMatter's tenant controllers and Kamiwaza's Ray sidecar present it when they connect to other mesh services, including GreyMatter Core. So it must:

- be issued by the CA that your GreyMatter Core trusts for mesh (PKI) traffic. That CA is the `ca.crt` of Core's PKI certificate Secret, by default `greymatter-edge-ingress` in the `greymatter` namespace. A certificate from any other CA, or a self-signed one, leaves the tenant controllers unable to connect to GreyMatter Core, and GreyMatter never applies Kamiwaza's configuration;
- be valid for both server and client authentication (extended key usages `serverAuth` and `clientAuth`).

Publicly trusted CAs don't issue certificates with client authentication, and browsers don't trust your mesh CA, so users see a certificate warning unless their machines trust that CA.

Create it in the Kamiwaza namespace, named `greymatter-edge-ingress`, with the server certificate and any intermediates in `tls.crt` and your mesh CA in `ca.crt`:

```
oc -n kamiwaza create secret generic greymatter-edge-ingress \
  --from-file=ca.crt=./edge-ca.crt \
  --from-file=tls.crt=./edge.crt \
  --from-file=tls.key=./edge.key
```

Kamiwaza workloads need the mesh CA to verify the edge, since it is not publicly trusted. Publish the CA certificate in the Kamiwaza namespace:

```
kubectl -n kamiwaza create configmap kamiwaza-edge-trust \
  --from-file=ca-certificates.crt=./edge-ca.crt
```

This ConfigMap holds public certificates only. Never put a private key in it. Kamiwaza reads it by name when it exists; no install value refers to it.

Check:

```
oc -n kamiwaza get secret greymatter-edge-ingress -o jsonpath='{.data.tls\.crt}' \
  | base64 -d | openssl x509 -noout -subject -issuer -ext subjectAltName -enddate -
purpose \
  | grep -E 'subject|issuer|DNS:|notAfter|^SSL (client|server) :'
# Expect your domain under subjectAltName, an end date in the future,
# "SSL client : Yes" and "SSL server : Yes"
oc -n kamiwaza get secret greymatter-edge-ingress -o json | jq -r '.data | keys |
join(" ")'
# Expect ca.crt tls.crt tls.key

# The certificate chains to the CA GreyMatter Core trusts
# (use your Core PKI Secret if it isn't the default):
oc -n greymatter get secret greymatter-edge-ingress -o jsonpath='{.data.ca\.crt}' \
  | base64 -d > greymatter-mesh-ca.crt
oc -n kamiwaza get secret greymatter-edge-ingress -o jsonpath='{.data.tls\.crt}' \
  | base64 -d > kamiwaza-edge.crt
openssl verify -CAfile greymatter-mesh-ca.crt -untrusted kamiwaza-edge.crt kamiwaza-
edge.crt
# Expect kamiwaza-edge.crt: OK
```

Once this Secret exists, the GreyMatter tenant controllers from Enroll the namespace (see online documentation) start. Check that they are ready:

```
oc -n kamiwaza get deployments
# Expect greymatter-tcm and the other tenant controllers ready
```

If they were already running with a different certificate, restart them so they load this one: `oc -n kamiwaza rollout restart deployment <name>` for each.

DNS

The Kamiwaza domain must resolve to the OpenShift router's external address, the same way as other passthrough Routes on the cluster:

- **For users:** your DNS points the domain at the router's load balancer. Find its address with:

  ```
  oc -n openshift-ingress get service router-default \
    -o jsonpath='{.status.loadBalancer.ingress[0].hostname}
  {.status.loadBalancer.ingress[0].ip}{"\n"}'
  ```

  A host name (as on AWS) needs a `CNAME` record; an IP address needs an `A` record. The router passes TLS through to the GreyMatter edge, so don't put a proxy or CDN that terminates TLS in front of the domain: publish the record as plain DNS.

- **Inside the cluster:** Kamiwaza services and apps call the platform at its public domain, so pods must resolve it and reach the router the same way. OpenShift's cluster DNS forwards to your upstream resolvers, so this usually needs nothing more.

The router routes on the TLS server name, so clients must connect using the Kamiwaza domain. The Route itself is created when Kamiwaza's GreyMatter configuration is published during the install; until then the router answers for the domain with its own `503` page.

Check from your workstation, and from a pod:

```
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/
# Any HTTP status proves DNS reaches the router. Before Kamiwaza is
# installed, expect 503.

(
export KUBECONFIG=kamiwaza-installer.kubeconfig
oc -n kamiwaza run kamiwaza-dns-check --restart=Never --image=curlimages/curl \
  -- -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/
oc -n kamiwaza wait --for=jsonpath='{.status.phase}'=Succeeded \
  pod/kamiwaza-dns-check --timeout=60s
oc -n kamiwaza logs kamiwaza-dns-check
# Expect the same status from inside the cluster
oc -n kamiwaza delete pod kamiwaza-dns-check
)
```

The pod commands run with the installer kubeconfig for the same reason as the block storage (see online documentation) write test: as cluster-admin, OpenShift can admit the pod under a security context constraint that assigns it no user ID.

Record: the domain.

Block storage

Kamiwaza's databases and caches use `ReadWriteOnce` persistent volumes. The cluster must have a StorageClass that provisions them. Record its name; the install names it explicitly.

Size the storage provider for Kamiwaza's volume requests plus your provider's replication and free-space reserves. A StorageClass existing does not prove its volumes can be provisioned; the write test below does. Node-local storage, such as `local-path`, works on a single node but does not replicate data.

Kamiwaza requests these volumes by default, about 624 GiB in total. Apps you deploy later add their own.

Volume	Default request	Holds	Change it with
`kamiwaza-registry-data`	500 GiB	Model and inference-engine cache	`registry.persistence.size`
`seaweedfs-data`	100 GiB	Built-in object store	`seaweedfs.persistence.size`
`core-postgres-data`	10 GiB	Platform database	—
`keycloak-postgres-data`	10 GiB	Identity database	—
`core-spicedb-postgres-data`	1 GiB	Authorization database	—
`core-etcd-data-core-etcd-0` to `-2`	1 GiB each	Configuration store	—

The model registry cache holds inference-engine images and the model files your users deploy. If you serve only external models (for example, through a cloud provider), or your storage is small, set `registry.persistence.size` lower, for example `100Gi`, in your install values before the first install. An existing volume keeps its size on upgrade.

Check the classes, then run a write test in the Kamiwaza namespace: write a marker, replace the pod, and read the marker back. Run the test with the installer kubeconfig from Installer identity (see online documentation), as below. As cluster-admin, OpenShift can admit the test pod under a security context constraint that assigns it no user ID, and the pod then fails to start.

```
kubectl get storageclass
# Expect one class marked (default), or note the class you will name at install

(
export KUBECONFIG=kamiwaza-installer.kubeconfig
cat > kamiwaza-storage-check.yaml <<'EOF'
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kamiwaza-storage-check
spec:
  accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 1Gi
  # storageClassName: <class>    # uncomment if you will not use the default
---
apiVersion: v1
kind: Pod
metadata:
  name: kamiwaza-storage-check
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: check
      image: busybox:1.37
      command: ["sh", "-c", "sleep 3600"]
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: ["ALL"]
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: kamiwaza-storage-check
EOF

kubectl -n kamiwaza apply -f kamiwaza-storage-check.yaml
kubectl -n kamiwaza wait --for=condition=Ready pod/kamiwaza-storage-check --timeout=5m
kubectl -n kamiwaza exec kamiwaza-storage-check -- sh -c 'echo ok > /data/marker'

# Replace the pod; the claim and its volume stay
kubectl -n kamiwaza delete pod kamiwaza-storage-check
kubectl -n kamiwaza apply -f kamiwaza-storage-check.yaml
kubectl -n kamiwaza wait --for=condition=Ready pod/kamiwaza-storage-check --timeout=5m
kubectl -n kamiwaza exec kamiwaza-storage-check -- cat /data/marker     # expect: ok

kubectl -n kamiwaza delete -f kamiwaza-storage-check.yaml
)
```

A class with `volumeBindingMode: WaitForFirstConsumer` binds the claim only once the pod is scheduled, so a `Pending` claim before the pod starts is normal.

Record: the StorageClass name.

Object storage

Workrooms, the Skills Library, and other features store objects. Choose one:

- **The built-in object store (default).** Kamiwaza runs SeaweedFS in its namespace, backed by a `ReadWriteOnce` volume from your block storage. Nothing to prepare.
- **Your own S3-compatible service.** See S3 workroom storage (see online documentation).

Kubernetes API addresses

Some Kamiwaza apps call the Kubernetes API from inside the namespace, and Kamiwaza allows that traffic with NetworkPolicy. Many CNIs, including the ones k0s and OpenShift use, evaluate NetworkPolicy **after** a Service address is translated to the API server's real address. So the policy needs both the `kubernetes` Service address **and** every API server endpoint address. With only the first, apps install cleanly and then fail at runtime.

While OpenShift rolls out the API server or restarts a control-plane node, for example on a new cluster or after a configuration change, the endpoint list leaves out the server being restarted. Record the addresses only when no rollout is in progress:

```
oc get clusteroperator kube-apiserver
# Expect PROGRESSING False. If the cluster has no kube-apiserver cluster
# operator, as can happen when the provider hosts the control plane, skip this.
oc get machineconfigpool master
# Expect UPDATED True and UPDATING False. Skip this too if the provider hosts
# the control plane.
```

Record both sets of addresses, as `/32` for IPv4 or `/128` for IPv6:

```
kubectl -n default get service kubernetes -o jsonpath='{.spec.clusterIPs[*]}{"\n"}'
kubectl -n default get endpointslices -l kubernetes.io/service-name=kubernetes \
  -o jsonpath='{.items[*].endpoints[*].addresses[*]}{"\n"}'
# Expect one endpoint address for each API server: three on a cluster with
# three control-plane nodes
```

Copy the addresses exactly. A wrong address still installs cleanly, and the apps then fail at runtime. If the API server's addresses change (for example, a control-plane node is replaced), update them in the install values and upgrade.

If your CNI enforces NetworkPolicy **before** Service address translation, the endpoint addresses are not needed; the Service address alone works. Record it anyway.

Registry access

Kamiwaza's charts and images are published to a public registry and pull without credentials. Nodes must be able to pull from it; see Outbound network access (see online documentation).

If you mirror images into your own registry and it requires credentials, create a `kubernetes.io/dockerconfigjson` pull Secret in the Kamiwaza namespace. Kamiwaza uses only namespaced pull Secrets, never node-wide registry credentials. Build the Secret from a file, so the password never appears in a command line or your shell history:

```
read -rs REGISTRY_PASSWORD      # typed, not echoed
(umask 077; d=$(mktemp -d)
 printf '{"auths":{"%s":{"auth":"%s"}}}' '<registry-host>' \
   "$(printf '%s:%s' '<username>' "$REGISTRY_PASSWORD" | base64 | tr -d '\n')" >
"$d/config.json"
 kubectl -n kamiwaza create secret generic kamiwaza-pull \
   --type=kubernetes.io/dockerconfigjson --from-file=.dockerconfigjson="$d/config.json"
 rm -rf "$d")
unset REGISTRY_PASSWORD
```

`<registry-host>` is the registry's host name, with its port if it has one, as it appears in the image references.

Record: the pull Secret's name, `kamiwaza-pull` above. The install lists it in `global.imagePullSecrets`.

Nodes must also pull the GreyMatter proxy image. GreyMatter provides its pull Secret; see Enroll the namespace (see online documentation).

Node kernel limits

Every GreyMatter sidecar watches files with inotify. Node defaults can be too low for the number of pods Kamiwaza runs, and sidecars then crash with `too many open files`. Set these floors on **every worker node**.

On OpenShift, set them with a `MachineConfig` for each machine pool that runs Kamiwaza. Whether a custom `MachineConfig` is supported depends on your provider's support policy. If your platform doesn't let you create one (for example, ROSA with hosted control planes), set the same limits through your provider's node tuning mechanism, or ask your provider's support to set them.

Red Hat Enterprise Linux CoreOS already sets `fs.inotify.max_user_watches` to 65536 in `/etc/sysctl.d/inotify.conf`. Files in `/etc/sysctl.d` apply in name order and the last one wins, so Kamiwaza's file must sort after that one; the `zz-` prefix below does that.

This `MachineConfig` covers the `worker` pool, which on ROSA also includes the infra nodes. Save it as `kamiwaza-inotify-machineconfig.yaml`:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  name: 99-worker-kamiwaza-inotify
  labels:
    machineconfiguration.openshift.io/role: worker
spec:
  config:
    ignition:
      version: 3.2.0
    storage:
      files:
        - path: /etc/sysctl.d/zz-kamiwaza-inotify.conf
          mode: 0644
          overwrite: true
          contents:
            source:
data:,fs.inotify.max_user_instances%20%3D%208192%0Afs.inotify.max_user_watches%20%3D%201048
```

```
oc apply -f kamiwaza-inotify-machineconfig.yaml
oc get machineconfigpool worker -w     # wait until UPDATED is True
```

Applying a `MachineConfig` restarts the pool's nodes one at a time, a few minutes each. Schedule it with your platform's change process. If Kamiwaza is already installed, expect a brief interruption while the node running its edge restarts.

Check on each worker node:

```
oc debug node/<node> -- chroot /host \
  sysctl fs.inotify.max_user_instances fs.inotify.max_user_watches
fs.inotify.max_queued_events
# Expect values at or above 8192, 1048576, and 16384
```

Outbound network access

Nodes and pods need outbound HTTPS (port 443) to:

- `ghcr.io` and `pkg-containers.githubusercontent.com`, for Kamiwaza charts and images;
- the model sources you use, for example `huggingface.co` and its content hosts, or your cloud model provider's endpoint;
- your Git server, for the tenant configuration repository (see online documentation);
- the registry that serves the GreyMatter proxy image.

The checks on this page use the public `busybox` and `curlimages/curl` images from Docker Hub. If your cluster cannot reach Docker Hub, mirror them to your registry.

For installs with no outbound access, see Install Without Internet Access (see online documentation).

GPU nodes

GPU inference is optional. To serve models on NVIDIA GPUs, prepare each GPU node as described in GPU Nodes (see online documentation): a driver that meets the engine images' floor, persistence mode, the NVIDIA container runtime and the `nvidia` RuntimeClass, the NVIDIA GPU Operator, and no `NoSchedule` or `NoExecute` taint on the node. Kamiwaza installs none of these. The node commands on that page are for Ubuntu nodes. On OpenShift, install the NVIDIA GPU Operator with your platform's usual method, then run the checks on that page that apply to your nodes.

Prepared nodes are not enough on their own: a default install cannot use GPUs. After the install, the cluster owner turns GPU serving on with Tenant-mode inference (see online documentation).

Check:

```
kubectl get nodes -o custom-
columns='NODE:.metadata.name,GPU:.status.allocatable.nvidia\.com/gpu'
# Expect a count on each GPU node
kubectl get runtimeclass nvidia
# Expect a RuntimeClass named nvidia, with handler nvidia
```

Record: which nodes have GPUs, and the GPU model on each. The cluster owner needs them to publish a signed profile.

Multi-node clusters

On a cluster with more than one node, also provide:

- **Storage that survives a node loss.** Use a StorageClass whose volumes are replicated or network-attached, such as a CSI driver for your SAN or cloud block storage, or a replicated provider like Longhorn. Node-local storage ties each database to one node: if that node is lost, so is the data.
- **Every GPU node prepared.** Prepare each GPU node as in GPU nodes (see online documentation), including nodes added later.
- **A CNI that enforces NetworkPolicy.** Kamiwaza isolates apps and jobs from each other with NetworkPolicy. A CNI that accepts policies but does not enforce them leaves that isolation off without any error.
- **The inotify floor on every node**, including nodes added later.
- **The API endpoint addresses of every control-plane node** in the recorded Kubernetes API addresses (see online documentation).

Next step

When every check passes, hand the recorded facts and the installer kubeconfig to whoever runs the install, and continue to Install on OpenShift with GreyMatter (see online documentation).

Install on OpenShift with GreyMatter

This page installs Kamiwaza into one namespace of an OpenShift cluster whose service mesh is GreyMatter. You publish Kamiwaza's GreyMatter configuration, then run a single `helm upgrade --install` as a namespace-scoped identity. The chart creates only namespaced objects.

Before you start

You need	Notes
A prepared cluster	Every check in Platform Prerequisites (see online documentation) passes, and you have the facts it told you to record.
The installer kubeconfig	The namespace-scoped identity from Installer identity (see online documentation). Never install with a cluster-admin kubeconfig.
`oc` or `kubectl`, and Helm 3.14 or later	Helm pulls the chart from an OCI registry. With Helm 4, add `--server-side=false` to every `helm upgrade --install` on this page; see step 4 (see online documentation).
`jq`, `curl`, and `git`	Used by the configuration and verification steps.
Push access to the tenant configuration repository	The repository and branch from Tenant configuration repository (see online documentation). If the platform administrator keeps that access, they run step 3 (see online documentation).
A license file	`license.lic`, issued by Kamiwaza. See License File (see online documentation).

No internet access? Do Install Without Internet Access (see online documentation) first. It copies the release into your registry and gives you values to merge into your values file in step 2. In steps 3 and 4, use the chart from your registry in place of the release location.

Point `kubectl` and `helm` at the installer identity for every command on this page:

```
export KUBECONFIG=<path to the installer kubeconfig>
```

Kamiwaza installs into the `kamiwaza` namespace. The examples use `kamiwaza.example.com` as the domain; substitute your own.

Step 1: Add the license

Create a Secret from the license file. The key must be `license.lic`:

```
kubectl -n kamiwaza create secret generic kamiwaza-license \
  --from-file=license.lic=./license.lic --dry-run=client -o yaml \
  | kubectl apply -f -
```

This also works on a reinstall: uninstalling keeps this Secret, and the command updates it in place.

The chart refuses to render until a license is configured and the EULA is accepted. You do both in the values file in the next step.

Step 2: Write your values file

Your values file selects the OpenShift with GreyMatter platform and carries this cluster's facts and your acceptance of the EULA. Everything else is the chart's defaults for that platform. Keep the file somewhere safe and outside any shared repository; you will need it again for upgrades.

Required values

Value	Set it to
`global.platform`	`osgm`. This switches the chart to GreyMatter routing and OpenShift's security model.
`global.eula.accepted`	`true`, after reading the EULA (https://www.kamiwaza.ai/eula-license). The render fails without it.
`core.license.existingSecret`	`kamiwaza-license`, the Secret from step 1. The render fails without it.
`global.domain`	The Kamiwaza domain.
`global.namespaces.platform`, `.extensions`, `.sandboxes`, `.observability`, `.ca`, `.registry`, `.system`	`kamiwaza`, for each one. Everything runs in that one namespace.
`global.storage.stateful.standard`, `.postgres`, `.etcd`, `.registry`, `.extensions`	The StorageClass you recorded, for each one.
`core.extensionRuntime.kubernetesApi.egressCidrs`	Every Kubernetes API Service address **and** every endpoint address you recorded, each as `/32` (IPv4) or `/128` (IPv6). Copy them exactly: a wrong address installs cleanly and then fails at runtime. The render fails without this value.
`global.greymatter.proxy.image`	The GreyMatter proxy image you recorded, split into `registry`, `repository`, and `tag`. It must match the proxy version your GreyMatter platform runs.

Values that depend on your cluster

Value	Set it when
`global.greymatter.proxy.imagePullSecret`	GreyMatter's image pull Secret in the namespace is not named `greymatter-image-pull`.
`global.imagePullSecrets`	Kamiwaza images come from a registry that needs credentials. List the pull Secret you created, for example `[{name: kamiwaza-pull}]`.
`core.inferenceResources.enabled`, `.bundleConfigMap`, `.ownerPublicKey`, `.clusterBindingId`	You serve models on GPUs. Set them after the cluster owner publishes a signed profile, as in Tenant-mode inference (see online documentation). Without them, a GPU deployment is refused. Prepare the nodes first; see GPU nodes (see online documentation).

Optional values

Value	Notes
`registry.persistence.size`, `seaweedfs.persistence.size`	Volume sizes for the model registry cache (default `500Gi`) and the built-in object store (default `100Gi`). Lower them when storage is small; see Block storage (see online documentation). Set them before the first install; they can't be reduced later.
`core.userDefaults.adminPassword`	The password for the `admin` user. Leave it unset and Kamiwaza generates one, which you read back in step 5 (see online documentation). Setting it here stores it in the Helm release. It applies at first install only; changing it later does not rotate the password.

Example

```yaml
global:
  platform: osgm
  domain: kamiwaza.example.com
  eula:
    accepted: true
  namespaces:
    platform: kamiwaza
    extensions: kamiwaza
    sandboxes: kamiwaza
    observability: kamiwaza
    ca: kamiwaza
    registry: kamiwaza
    system: kamiwaza
  storage:
    stateful:
      standard: gp3-csi
      postgres: gp3-csi
      etcd: gp3-csi
      registry: gp3-csi
      extensions: gp3-csi
  greymatter:
    proxy:
      image:                                    # the proxy image you recorded
        registry: oci.download.greymatter.io
        repository: greymatter-proxy
        tag: 1.17.2

core:
  license:
    existingSecret: kamiwaza-license
  extensionRuntime:
    kubernetesApi:
      egressCidrs:
        - 172.30.0.1/32       # kubectl -n default get service kubernetes
        - 10.0.1.10/32        # the kubernetes EndpointSlice addresses
        - 10.0.1.11/32
        - 10.0.1.12/32
```

Save it as `kamiwaza-values.yaml`.

Step 3: Publish the GreyMatter configuration

GreyMatter needs Kamiwaza's configuration before the install. It defines the edge that serves the Kamiwaza domain, the routes to Kamiwaza's services, and the sidecars GreyMatter adds to them. The chart generates this configuration from your values file, so it always matches the release you install. You render it, commit it to the tenant configuration repository, and GreyMatter Sync applies it.

1. Clone the tenant configuration repository on its branch, and go to the project the platform administrator created:

```
git clone -b tenant/kamiwaza https://git.example.com/org/kamiwaza-gsl.git
kamiwaza-gsl
cd kamiwaza-gsl/_tenants/kamiwaza
```

If you already have a clone, run `git switch tenant/kamiwaza` and `git pull` in it instead.

2. Render the configuration and write its files into the project. This does not change the cluster.

```
helm template kamiwaza \
  oci://ghcr.io/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza \
  --version 1.3.4 \
  --namespace kamiwaza \
  --values <absolute path to kamiwaza-values.yaml> \
  --show-only charts/network/templates/greymatter/gsl-bundle.yaml \
  | kubectl create --dry-run=client --validate=false -f - -o json > gsl-
bundle.json

jq -r '.data["file-list.txt"]' gsl-bundle.json | grep '^greymatter/' \
  | while read -r f; do
    case "$f" in greymatter/generated/*) [ -e "$f" ] && continue ;; esac
    mkdir -p "$(dirname "$f")"
    jq -r --arg k "${f//\//__}" '.data[$k]' gsl-bundle.json > "$f"
  done
rm gsl-bundle.json
```

This writes the `greymatter/` directory. It leaves the `.greymatter` file and `cue.mod/` directory that the GreyMatter CLI created unchanged. It creates the files under `greymatter/generated/` only if they do not exist: Kamiwaza records its app and model routes there while it runs, so an upgrade must keep them.

3. Check that the configuration names your domain, then commit and push:

```
grep -rh 'api_endpoint' greymatter | sort -u
# Expect https://kamiwaza.example.com, and paths under it
git add greymatter
git diff --cached --quiet || git commit -m "Kamiwaza 1.3.4 GreyMatter
configuration"
git pull --rebase
git push
```

On a reinstall or an upgrade that changes nothing here, there is nothing to commit, and the push is a no-op.

Check that GreyMatter applied it and the edge serves the domain:

```
kubectl -n kamiwaza get route \
  -o custom-columns='NAME:.metadata.name,HOST:.spec.host,TLS:.spec.tls.termination'
# Expect a Route for kamiwaza.example.com with TLS "passthrough"
kubectl -n kamiwaza get pods -l app.kubernetes.io/name=edge
# Expect the edge pod Running and ready

curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/
# Any HTTP status from the edge proves the Route and edge certificate work.
# Before Kamiwaza is installed, expect 404 or 503.
```

GreyMatter can take a few minutes to sync. A connection that fails, or ends during the TLS handshake, is not a sync delay: see Troubleshooting (see online documentation).

Step 4: Install

Record the start time first. The verification step uses it to ignore events from before this install.

```
INSTALL_START=$(date -u +%Y-%m-%dT%H:%M:%SZ)

helm upgrade --install kamiwaza \
  oci://ghcr.io/kamiwaza-ai/releases/kamiwaza/charts/kamiwaza \
  --version 1.3.4 \
  --namespace kamiwaza \
  --values <absolute path to kamiwaza-values.yaml> \
  --timeout 70m
```

- Do not add `--create-namespace`. The namespace belongs to the platform administrator.
- With Helm 4, add `--server-side=false`, on the first install and every upgrade. OpenShift adds its own pull Secret to each of Kamiwaza's service accounts, and Helm 4's default server-side apply then refuses every later upgrade with `conflict with "openshift.io/image-registry-pull-secrets_service-account-controller"`. Helm 3 is not affected.
- Helm prints `WARNING: core cannot see GPUs on this install.` until tenant-mode inference is on. It is expected; see Tenant-mode inference (see online documentation).
- A typical install takes 5 to 15 minutes, most of it pulling images.

Install and upgrade with Helm, not rendered manifests

Always install and upgrade with `helm upgrade --install` against the cluster. Rendering the chart with `helm template` or `--dry-run` and applying the output, for example through Argo CD or Flux, is not supported. On first install the chart generates the platform's encryption keys, signing keys, and database passwords, and on each upgrade it reads them back from the cluster to keep them. A rendered chart cannot read the cluster, so every render generates new ones. Applying them makes existing encrypted data and access tokens unreadable.

Step 3 renders only the GreyMatter configuration files, which contain no generated secrets, and applies nothing to the cluster.

To upgrade to a later release in the same line, repeat step 3 with the new version, then rerun the command in this step with the new `--version` and the same values file.

Step 5: Verify

Check what the platform does, not only its status fields.

Wait for the platform to start. Helm can return before every pod is ready and every setup job has finished:

```
kubectl -n kamiwaza wait deployment -l app.kubernetes.io/managed-by=Helm \
  --for=condition=Available --timeout=30m
kubectl -n kamiwaza wait job -l app.kubernetes.io/managed-by=Helm \
  --for=condition=Complete --timeout=70m
```

Pods are running and ready:

```
kubectl -n kamiwaza get pods --no-headers \
  | awk '$3 != "Completed" { split($2, r, "/"); if ($3 != "Running" || r[1] != r[2])
print }'
# Expect no output
```

An `Error` pod whose Job later completed is a retried attempt, not a failure. Confirm with `kubectl -n kamiwaza get jobs`: every Job should show all completions.

GrevMatter sidecars are in place. The chart adds one to the `core-raycluster` pods; GreyMatter adds one to `frontend` and `keycloak`. A pod without it cannot receive traffic from the edge.

```
kubectl -n kamiwaza get pods \
  -o custom-columns='POD:.metadata.name,INIT:.spec.initContainers[*].name' \
  | grep -E '^(core-raycluster|frontend|keycloak)-' | grep -v '^keycloak-postgres'
# Expect greymatter-sidecar listed for each pod
```

Every pod runs under OpenShift's default security context constraint:

```
kubectl -n kamiwaza get pods \
  -o jsonpath='{range .items[*]}{.metadata.name}
{.metadata.annotations.openshift\.io/scc}{"\n"}{end}' \
  | grep -v -E ' restricted-v2$' | grep -v -E '^greymatter-'
# Expect no output
```

GreyMatter's own controllers in the namespace (`greymatter-*`) use GreyMatter's SCCs and are left out.

No image pulls failed during this install. Older events in the namespace are ignored. Run this in the same shell as step 4, or set `INSTALL_START` to the time you started the install, in the same format:

```
kubectl -n kamiwaza get events -o json | jq --arg start "$INSTALL_START" '
  [.items[]
   | select((.lastTimestamp // .eventTime // .metadata.creationTimestamp) >= $start)
   | select((.message // "") | test("ErrImagePull|ImagePullBackOff|Failed to pull
image|Back-off pulling image"))
  ] | length'
# Expect 0
```

The platform answers at its domain:

```
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/       # expect
200
curl -sk -o /dev/null -w '%{http_code}\n' https://kamiwaza.example.com/api/   # expect
401
```

Log in. If you did not set an admin password, read the generated one:

```
kubectl -n kamiwaza get secret kamiwaza-user-admin \
  -o jsonpath='{.data.password}' | base64 -d; echo
```

Open `https://kamiwaza.example.com/` and log in as `admin`. With a certificate your browser does not trust, accept the warning first.

`admin` is the Kamiwaza application administrator. It is not the installer identity and not the Keycloak management account.

Call the API. Exchange the same credentials for a token, and use it as a bearer token:

```
TOKEN=$(kubectl -n kamiwaza get secret kamiwaza-user-admin -o
jsonpath='{.data.password}' \
  | base64 -d | curl -sk https://kamiwaza.example.com/api/auth/token \
      --data-urlencode username=admin --data-urlencode password@- | jq -r
.access_token)
curl -sk -o /dev/null -w '%{http_code}\n' -H "Authorization: Bearer $TOKEN" \
  https://kamiwaza.example.com/api/auth/users/me      # expect 200
```

Step 6: Deploy the default embedding model

Kamiwaza ships a default embedding model, `all-MiniLM-L6-v2`, inside the release and adds it to the model catalog during the install. On OpenShift with GrevMatter it does not deploy it for you. Until you do, embedding requests fail with HTTP 503 and the code `embedding_unavailable`, and features that embed text, such as semantic search in documents, are unavailable. Deploying it needs no internet access.

1. Log in as `admin` and open **Models**.
2. In the `all-MiniLM-L6-v2` row, click **Deploy**.
3. Select the `embedding` configuration and click **Deploy**.

Deploy it once. The model's first pod can start before GreyMatter has added its sidecar to the model; Kamiwaza replaces that pod with one that has the sidecar, usually within a minute.

Check that the embedding answers, using the `TOKEN` from step 5 (rerun the step 5 command that sets it if the token has expired):

```
curl -sk -X POST -H "Authorization: Bearer $TOKEN" -H 'Content-Type: application/json'
\
  -d '{"text":"embedding check"}' https://kamiwaza.example.com/api/embedding/generate \
  | jq 'if .embedding then (.embedding | length) else . end'
# Expect 384. Otherwise the response is printed: the code embedding_deploying
# means the model is still starting; retry after a minute.

kubectl -n kamiwaza get pods \
  -o custom-columns='POD:.metadata.name,INIT:.spec.initContainers[*].name' | grep
'^llamacpp-'
# Expect one pod, with greymatter-sidecar listed. Within the first minute you
# can also see the replaced pod terminating.
```

The pod check applies with tenant-mode inference off, which is the default. With it on, model pods are named `tmi-<id>`; see Tenant-mode inference (see online documentation).

The platform is installed. Deploying models and apps is covered in the Quickstart (see online documentation).

Troubleshooting

The install stops before creating anything. The chart checks its values before rendering. A missing EULA acceptance or license prints `INSTALLATION HALTED`; a missing `egressCidrs` value names the value it needs. Nothing was installed. Fix the values file and rerun step 4.

`no Kamiwaza license is configured` with the license set. The license value belongs under `core:`. Check that it is `core.license.existingSecret`, not `license.existingSecret`. To see values Helm received but no chart reads, run `helm get values kamiwaza -n kamiwaza`.

The domain gives a TLS error, or no response, after step 3. Check that the Route's TLS termination is `passthrough`, that the edge pod is ready, and that `greymatter-edge-ingress` holds `tls.crt` and `tls.key` for the domain. An admitted Route with a ready edge still fails the handshake when the edge cannot read its certificate.

No Route or edge pod appears after step 3. GreyMatter has not applied the configuration. Ask the platform administrator to check that the namespace is enrolled, that the tenant controllers (for example `greymatter-tcm`) are ready, and that `greymatter-tenant-repo` names the repository, branch, and `_tenants/kamiwaza` path you pushed to. If `greymatter-tcm` is running but its log repeats `Failed to connect to a NATS cluster`, the edge certificate isn't issued by the mesh CA; see Edge certificate (see online documentation).

Pods do not start: `container has runAsNonRoot and image has non-numeric user`, or `image will run as root`. The pod was admitted under an SCC other than `restricted-v2`. See which with `kubectl -n kamiwaza get pod <pod> -o jsonpath='{.metadata.annotations.openshift\.io/scc}'`, and ask the

platform administrator to remove the extra SCC from Kamiwaza's service accounts, as in Security context constraints (see online documentation). Then delete the pod so it is recreated.

Pods stay in `CreateContainerConfigError`, naming `greymatter-tenant-repo`. The Secret is missing or lacks a key. `core-scheduler` and `core-raycluster` need `url`, `branch`, `relative_path`, `http_username`, and `http_password`.

The `core-raycluster` pods cannot pull the GreyMatter proxy image. Check `global.greymatter.proxy.image` against the image you recorded, and that the pull Secret exists: `kubectl -n kamiwaza get secret greymatter-image-pull`.

The install hangs for about 15 minutes, then fails on a post-install job. Check `core-scheduler` first: `kubectl -n kamiwaza logs deploy/core-scheduler`. A license problem shows here as `License check failed`; see License File (see online documentation). If `frontend` or `keycloak` are not ready, GreyMatter has not given them sidecars; check step 3. While the job runs, every `helm upgrade` fails with `another operation (install/upgrade/rollback) is in progress`.

Pods are `Pending` on their volumes. A StorageClass named in the values does not exist or cannot provision. Compare `kubectl get storageclass` with `global.storage.stateful.*`.

Images fail to pull with `unauthorized`. The pull Secret named in `global.imagePullSecrets` does not exist in the namespace, or has the wrong credentials. Compare it with `kubectl -n kamiwaza get secret`.

Apps install but cannot reach the Kubernetes API. `egressCidrs` is missing the endpoint addresses. Add every address from Kubernetes API addresses (see online documentation) and rerun step 4.

An app returns "Page Not Found" or the platform's own page, or a model's endpoint returns `no healthy upstream`. Kamiwaza adds each app's and model's route to the tenant configuration repository, and GreyMatter then applies it. Wait a minute or two after its pods are ready, or after the model shows as deployed. If it stays unreachable, check the `core-scheduler` logs for Git errors: the branch must accept direct pushes, and the namespace must reach the Git server.

Embedding requests return HTTP 503 with `embedding_unavailable`. No embedding model is deployed. Deploy the default one as in step 6 (see online documentation).

A GPU model deployment fails with HTTP 422 and `Core cannot see a GPU on this install`. Tenant-mode inference is off, which is the default. Prepare the GPU nodes, then turn it on; see Tenant-mode inference (see online documentation).

Next steps

- Quickstart (see online documentation): start using Kamiwaza.
- License File (see online documentation): check and rotate the license.
- Uninstall (see online documentation).

Uninstall from OpenShift with GreyMatter

This page removes a Kamiwaza installation from the `kamiwaza` namespace. It does not change the platform around it: the namespace, GreyMatter and Kamiwaza's configuration in your tenant configuration repository, storage classes, DNS, and certificates belong to the platform administrator and stay in place.

> **This is destructive.** Uninstalling deletes Kamiwaza's databases, stored objects, and model files along with the platform. Back up anything you need before you begin.

Step 1: Back up

Export or copy any data you need to keep, such as documents in workrooms, configuration, and anything in the platform databases. Step 3 deletes the platform databases and object store, and they cannot be recovered after it.

Step 2: Remove apps and models

Remove what you deployed through Kamiwaza while the platform is still running:

1. Log in as `admin`.
2. In the App Garden, stop every deployed app. In the Tool Shed, stop every tool server.
3. On the Models page, stop every model deployment, including the embedding model Kamiwaza deploys on its own, and remove any model provider connections you added.
4. Confirm that all three lists are empty.

Step 3: Uninstall the release

Run as the installer identity. This deletes the volume claims for the platform databases and the object store; step 4 describes what happens to the disks behind them:

```
helm uninstall kamiwaza --namespace kamiwaza --wait --timeout 15m
```

Step 4: Delete the remaining volumes

Two kinds of volume outlive the release: the model registry cache, which the chart keeps on purpose, and the configuration store's volumes, which Kubernetes keeps for any StatefulSet. GreyMatter shares this namespace, so delete Kamiwaza's claims by name rather than every claim in it:

```
kubectl -n kamiwaza get persistentvolumeclaims
kubectl -n kamiwaza delete persistentvolumeclaims --ignore-not-found \
  kamiwaza-registry-data core-etcd-data-core-etcd-0 core-etcd-data-core-etcd-1 core-
etcd-data-core-etcd-2
```

Whether the underlying disks are also deleted depends on the StorageClass's `reclaimPolicy`. With `Retain`, the platform administrator deletes the released PersistentVolumes.

Step 5: Verify

Run as the platform administrator, since the installer identity may not be able to list every kind:

```
kubectl api-resources --verbs=list --namespaced -o name \
  | grep -v -E '^events(\.events\.k8s\.io)?$' \
  | xargs -n 1 kubectl -n kamiwaza get --ignore-not-found -o name 2>/dev/null \
  | grep -v -E '^(packagemanifest|podmetrics)\.'
```

The last line drops kinds that OpenShift lists in every namespace: operator catalog entries and pod metrics.

Expect only the platform prerequisites, what OpenShift creates in every project, and what GreyMatter runs for the namespace:

- the installer's ServiceAccount, Roles, and RoleBindings;
- the `kamiwaza-license` Secret, and your pull Secret if you use one;
- the `greymatter-tenant-repo`, `greymatter-edge-ingress`, and `greymatter-image-pull` Secrets;
- the `kamiwaza-edge-trust` ConfigMap;
- the `kube-root-ca.crt` and `openshift-service-ca.crt` ConfigMaps, and the service accounts, Secrets, and RoleBindings OpenShift adds to every project;
- GreyMatter's tenant controllers, the edge and its Route, and their GreyMatter resources. GreyMatter keeps these while Kamiwaza's configuration is in the tenant configuration repository. They include the `edgeresources` and `sidecarresources` objects, the `greymatter-sidecar-injection-certs` Secret, the `greymatter-sidecar-injection-service` Service, the `greymatter-cni-greymatter` NetworkAttachmentDefinition, and the controllers' leases.

If anything else remains, contact Kamiwaza support with the list before reinstalling into the same namespace.

Removing the platform prerequisites

If Kamiwaza will not be installed again, the platform administrator can remove what was prepared for it, following their own procedures for GreyMatter, storage, DNS, and certificates. Work in this order:

1. Remove Kamiwaza's configuration from the tenant configuration repository (`_tenants/kamiwaza`), and let GreyMatter Sync apply the change. GreyMatter then removes the edge and its Route.

2. Remove the namespace from `tenant_namespaces` (and `ftr_namespaces`, if you added it) in your GreyMatter Core configuration.
3. Delete the `kamiwaza` namespace, which removes the installer identity, the Secrets, and everything else in it:

   ```
   oc delete project kamiwaza
   ```

4. Remove the DNS records for the Kamiwaza domain. If nothing else relies on the inotify limits, remove them the way you set them: delete the `99-worker-kamiwaza-inotify` `MachineConfig`, which restarts the pool's nodes, or, where you couldn't use a `MachineConfig`, undo the setting the way you applied it.

To install again after deleting the namespace, start from Platform Prerequisites (see online documentation).

Configuration Reference

Current Kamiwaza deployments are Kubernetes-based. In customer environments, configuration should be managed through your Helm values and cluster release workflow, not through ad hoc edits to source-managed `env.sh` files.

Configuration Model

In the current deployment model, configuration comes from three main layers:

- Helm values and override files in the deployment workflow
- Kubernetes ConfigMaps and Secrets rendered from those values
- a small set of runtime-managed settings stored through platform APIs

For most teams, the best practice is:

1. set customer-specific defaults in Helm values
2. keep secrets in Kubernetes Secrets or an external secret manager
3. use the platform API only for runtime settings that are explicitly designed to be changed after deployment

Start With These Docs

Use these guides together:

- System Requirements (see online documentation)
- Installing Kamiwaza (see online documentation)
- Database Schema Lifecycle (see online documentation)
- Quickstart (see online documentation)
- Administrator Guide (see online documentation)
- AWS S3 Workroom Storage (see online documentation)
- Observability (see online documentation)

Core Platform Settings

Domain and Origin

The most important public-access setting is the deployment domain.

In the Kubernetes deployment charts, this is typically set through:

```
global:
  domain: kamiwaza.example.com
```

That domain is then used to derive platform-facing values such as:

- `KAMIWAZA_EXTERNAL_URL`
- `KAMIWAZA_ORIGIN`
- allowed browser origins for authenticated platform traffic

Best practice:

- set `global.domain` to the canonical customer-facing hostname
- keep `KAMIWAZA_ORIGIN` aligned to that same HTTPS origin
- avoid mixing multiple “primary” hostnames unless your ingress and auth setup explicitly require it

Authentication and Security Configuration

Authentication, identity provider integration, consent, banners, ReBAC, and secrets management are covered in the Administrator Guide (see online documentation).

Best practice:

- manage auth and banner settings through your Helm values and platform admin workflow
- keep secrets out of plain-text values files whenever possible
- use Kubernetes Secrets, sealed secrets, or your cluster’s secret-management pattern for credentials and keys

If you are deploying a federal CAC-enabled environment, also use:

- CAC Overview (see online documentation)

Object Storage

If users will upload files, preserve workroom context, or use the Skills Library, object storage must be available before rollout.

Default self-managed installs use namespace-local SeaweedFS for workroom object storage; no object-store configuration is needed. Its volume uses the block storage class you name at install; see Block storage in your environment's prerequisites: Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation). When workroom content must live in external AWS S3, use:

- AWS S3 Workroom Storage (see online documentation)

Best practice:

- set bucket, region, and secret references in your Helm values file, under `core.context.objectStorage`

- prefer secret references or ambient cloud identity over inline credentials

Logging and Observability

Logging and telemetry configuration should be treated as platform configuration, not per-user customization.

Use:

- Observability (see online documentation)

Best practice:

- verify the UI log viewer works for deployment troubleshooting
- configure any OpenTelemetry or external log forwarding through your platform release values
- validate log access and retention as part of rollout readiness

CORS and Browser Origins

Kamiwaza supports additional browser origins through serving configuration, including `KAMIWAZA_CORS_ORIGINS`.

Best practice:

- keep browser origins specific
- do not use wildcard CORS unless you are in a controlled development environment
- make sure the configured allowed origins match the actual HTTPS origins users and extensions will access

If credentials are in use, broad or mismatched CORS settings are a common source of extension and browser failures.

Routing Configuration

Routing is now managed as a runtime control-plane setting, backed by the shared runtime store rather than per-pod file edits.

The routing API is:

- `GET /api/config/routing`
- `PATCH /api/config/routing`

Current best practice is to treat path-based routing as the standard customer-facing mode.

Canonical runtime prefixes are:

- models: `/runtime/models/<deployment-id>`
- apps: `/runtime/apps/<deployment-id>`

- tools: `/runtime/tools/<deployment-id>`

The Kubernetes charts already seed path-based routing defaults. Use the routing API only when you intentionally need to adjust the base host or service prefixes after deployment.

For more detail, see:

- Routing & URLs (see online documentation)

Where To Change Settings

Use this table as a quick guide:

Setting Area	Best Place to Change It
Customer hostname / ingress domain	Helm values
Auth and IdP integration	Helm values + admin workflow
Secrets and credentials	Kubernetes Secrets or external secret manager
Object storage	Helm values + secret references
Observability endpoints and log forwarding	Helm values
Runtime routing prefixes / base host	Routing API

Anti-Patterns to Avoid

- editing source-repo install scripts as a customer configuration mechanism
- treating old package-install `env.sh` flows as the primary deployment model
- storing cloud credentials directly in user-editable metadata
- documenting port-specific runtime URLs as the preferred public access pattern

Validation Checklist

After a configuration change, validate:

- the platform is reachable at the expected HTTPS origin
- administrator and standard-user sign-in still work
- app and tool launches resolve under the expected runtime paths
- storage-backed workflows still function
- logs remain visible to the right administrators

When in doubt, use the Quickstart (see online documentation) as the post-change validation path.

Database Schema Lifecycle

Kamiwaza v1.0 adds a bounded database schema anchor for PostgreSQL-backed installs. This is a safety foundation for future migrations, not the full migration platform.

The v1.0 contract is:

- Fresh supported installs initialize the database through the normal deployment path.
- The core database is stamped as the supported v1.0 schema floor.
- Startup fails closed if the database advertises an unknown or newer schema version.
- Downgrading a stamped database below v1.0 is not supported.
- Full migration runner behavior is deferred to the v1.1 migration platform.

Supported Initialization Path

For PostgreSQL-backed Kubernetes installs, the supported initialization gate is the `core-db-init` Helm hook. The current chart runs the core image initializer at `/app/services/core/scripts/db-init.py` before scheduler startup. Older versioned documentation retains the path used by those historical release artifacts.

During v1.0 initialization, the initializer creates or verifies the `kamiwaza_schema_version` table and records the core schema marker:

Field	Expected value
`schema_name`	`core`
`version`	`1.0`

The stamp operation is idempotent. Re-running the supported init path against a database already marked as `core=1.0` should leave the marker in place and continue.

Baseline Snapshot

v1.0 also includes a frozen, PostgreSQL-compatible baseline snapshot for the core database schema. The snapshot is the canonical floor that future v1.1 migration tooling can use without reconstructing all pre-v1.0 schema history.

The baseline covers tables owned by the v1.0 core initialization path, including the schema marker table. Lite-mode SQLite metadata is excluded from this PostgreSQL baseline.

Normal post-v1.0 schema changes should not rewrite the accepted v1.0 baseline. Treat changes after the baseline as migration work for the migration platform.

Failure Behavior

Kamiwaza must not blindly run schema initialization against a database owned by a newer or unrecognized binary. If the database contains a `kamiwaza_schema_version` row for `core` with a version other than `1.0`, v1.0 treats that state as unsupported and fails startup or initialization with an actionable error.

Common meanings:

Database state	v1.0 behavior
No marker on a fresh supported install	Initialize schema and stamp `core=1.0`
Existing `core=1.0` marker	Continue; the marker is known
Existing `core` marker with a future or unknown version	Fail closed; do not run blind initialization

If a deployment fails because the database is ahead of the running binary, do not downgrade the application onto that database. Restore the matching application version, restore from a compatible backup, or contact Kamiwaza support for the correct recovery path.

Downgrades

Downgrading a database stamped by v1.0 to an application version below v1.0 is unsupported. Pre-v1.0 binaries do not understand the v1.0 schema anchor and should not be used as a rollback target for a stamped PostgreSQL database.

For rollback planning, keep backups aligned with the application release you are rolling back to. A database backup taken after the v1.0 stamp should be restored only with a version that understands that schema state.

What Moves to v1.1

The v1.0 schema anchor is intentionally narrow. These items are not part of the v1.0 database lifecycle contract:

- full Alembic migration runner
- v1.0-to-v1.1 migration scripts
- downgrade or rollback migration tooling
- broad migration rehearsal CI
- operator live-upgrade runbook
- schema status API
- SQLite/lite-mode migration platform coverage

Those items belong to the v1.1 database migration platform.

Operator Checks

After install or upgrade, operators can verify the anchor by inspecting the core PostgreSQL database for the marker row:

```
SELECT schema_name, version
FROM kamiwaza_schema_version
WHERE schema_name = 'core';
```

The expected v1.0 result is:

```
schema_name | version
core        | 1.0
```

If the table or row is missing after a supported PostgreSQL install has completed successfully, collect the `core-db-init` job logs and contact Kamiwaza support before retrying with ad hoc database changes.

Routing Modes and URLs

Current Kamiwaza deployments use HTTPS ingress with path-based runtime URLs as the standard public access pattern.

In Kubernetes deployments, the charts seed path-based routing by default for:

- models
- apps
- tools

Canonical Runtime Paths

The standard public URL shape is:

- Models: `/runtime/models/<deployment-id>`
- Apps: `/runtime/apps/<deployment-id>`
- Tools: `/runtime/tools/<deployment-id>`

Examples:

```
https://kamiwaza.example.com/runtime/models/<deployment-id>
https://kamiwaza.example.com/runtime/apps/<deployment-id>
https://kamiwaza.example.com/runtime/tools/<deployment-id>
```

For model runtimes that expose OpenAI-compatible APIs, the model-scoped base URL follows the same pattern:

```
https://kamiwaza.example.com/runtime/models/<deployment-id>/v1
```

Best Practice

For customer-facing documentation, integrations, bookmarks, and extension guidance:

- use path-based URLs as the canonical form
- treat the platform HTTPS origin as the stable entrypoint
- avoid documenting port-specific runtime URLs as the preferred access pattern

This aligns with the current Kubernetes deployment model. Istio is the default routing provider. Greymatter is the supported alternative on deployments where Greymatter Core is installed and managed outside the Kamiwaza chart.

(Note: In local development deployments using the `k0s-podman` or `k0s-lima` Mesh M1 configuration, TLS is handled via a self-signed placeholder certificate. For production, see Production Certificate Swaps (see online documentation).)

Production Certificate Swaps

When deploying Kamiwaza in a production environment with Mesh M1, you must replace the default placeholder certificate with a valid TLS certificate for your domain.

The typical procedure involves:

1. **Configuring DNS:** Ensure your target domain (e.g., `kamiwaza.example.com`) resolves to the external IP of the Istio Ingress Gateway.
2. **Using cert-manager:** Deploy `cert-manager` to automatically provision and rotate certificates from Let's Encrypt or your internal CA.
3. **Updating the Istio Gateway secret:** Configure your Istio `Gateway` resource to reference the TLS secret generated by `cert-manager` (or manually provisioned) so that the ingress terminates TLS correctly.

Routing Configuration API

Routing is managed through the platform configuration API:

- `GET /api/config/routing`
- `PATCH /api/config/routing`

The API supports:

- `base_host`
- `service_prefixes`
- `enabled_services`

In the current implementation, the API reports and persists routing in `path` mode for the public runtime paths.

Default Kubernetes Routing Behavior

The deployment charts seed routing with path-based defaults similar to:

```
{
  "routing": {
    "mode": "path",
    "model_routing_method": "path",
    "base_host": "https://<platform-origin>",
    "service_prefixes": {
      "api": "/api",
      "frontend": "/",
      "models": "/runtime/models",
      "apps": "/runtime/apps",
      "tools": "/runtime/tools",
      "notebooks": "/notebooks",
      "admin_ray": "/admin/ray"
    }
  }
}
```

The customer-facing hostname is typically derived from the deployment domain, and the scheduler config maps that into:

- `KAMIWAZA_EXTERNAL_URL`
- `KAMIWAZA_ORIGIN`

Best practice:

- keep your canonical public hostname aligned with `global.domain`
- keep `KAMIWAZA_ORIGIN` aligned to the same HTTPS origin
- only change routing prefixes when you have a clear platform-wide reason to do so

Legacy Port-Based Routing

Some code paths still retain compatibility for legacy port-based or dual-mode routing behavior. That exists for backward compatibility, internal transitions, or older integrations.

For public docs and new integrations, path-based routing should be considered the supported standard unless your deployment team has explicitly documented a different compatibility requirement.

Migrating from Traefik

Traefik routing support was removed after the platform moved to Istio and Envoy `ext_authz`. An installation that explicitly sets `KAMIWAZA_ROUTING_PROVIDER=traefik` now fails during startup with migration guidance instead of silently selecting another provider.

To migrate an older installation:

1. Remove the `traefik` provider override and any `network.traefik` values.
2. For chart installs, set `global.mesh.provider=istio` and `global.ingress.provider=istio`. Select `greymatter` in both keys only when the cluster has a supported Greymatter Core installation. The chart renders `KAMIWAZA_ROUTING_PROVIDER`; set that environment variable directly only for a non-chart

process launch.

3. Replace custom Traefik `IngressRoute` and `Middleware` resources with the route mechanism owned by the selected provider.
4. Render the deployment values and verify the canonical runtime paths before upgrading production traffic.

Do not translate Traefik ForwardAuth headers into application configuration. Both surviving providers carry the authoritative request path in `:path`.

App and Tool URL Behavior

App and tool deployments use the same runtime path model:

- apps open under `/runtime/apps/<deployment-id>`
- tools open under `/runtime/tools/<deployment-id>`

The built-in Workrooms page and App Garden experiences rely on these path-based URLs when launching runtime apps.

Administrative and Platform Paths

Not every internal or administrative surface is meant to be documented as a public runtime path.

The main customer-facing patterns to document are:

- `/api`
- `/runtime/models/...`
- `/runtime/apps/...`
- `/runtime/tools/...`

Additional administrative paths such as `/admin/ray` are platform-specific and should be documented only when they are intentionally exposed in a given environment.

When To Update Routing

Use the routing API when you need to:

- move the canonical base host to a different customer-facing domain
- change a service prefix intentionally across the platform
- disable or hide a specific routed service prefix

Do not treat pod-local config files such as `runtime_config.json` as the primary manual control point for customer operations. In current deployments, the shared runtime configuration and Helm-rendered defaults are the source of truth.

Validation Steps

After a routing change, validate:

1. the platform home page loads at the intended HTTPS origin
2. `/api/ping` or another known API endpoint resolves correctly
3. at least one model URL works under `/runtime/models/<deployment-id>/v1`
4. at least one App Garden app opens under `/runtime/apps/<deployment-id>`
5. auth redirects and logout flows still return users to the correct origin

Use the Quickstart (see online documentation) and Observability (see online documentation) guides if a routing change breaks app launches or browser flows.

AWS S3 Workroom Storage

Kamiwaza uses S3-compatible object storage for two platform workflows:

- workroom context file persistence
- Skills Library package import and download

If object storage is not configured, file-backed user workflows can fail with errors such as:

```
Workroom storage is not configured for Skills Library.
```

Use this guide when your deployment stores workroom content in **external AWS S3** instead of the default in-cluster object storage.

> **⚠ DEFAULT SEAWEEDFS INSTALLS NEED NO CONFIGURATION — DO NOT APPLY THIS PAGE TO THEM**
>
> By default the chart deploys SeaweedFS in the `kamiwaza` namespace and wires workroom storage to it. On those installs workroom storage already works, and applying this page's values without a real external-S3 requirement — or leaving them behind after one — will break it.
>
> In particular, a `core.context.objectStorage` block left in your values file from an earlier external-S3 setup replaces the built-in wiring, and core then references a Secret that may not exist. The install never completes: `helm upgrade --install` waits on post-install jobs while core pods sit in `CreateContainerConfigError: secret "core-s3" not found`. See Troubleshooting (see online documentation) for diagnosis and cleanup.

When You Need This

Configure AWS S3 if your workroom content must live in an external AWS S3 bucket and you want the following to work reliably:

- context file upload and persistence
- Skills Library package import
- Skills Library package download

If the default SeaweedFS backend meets your requirements, you do not need this page.

Prerequisites

Before you start, make sure you have:

- an AWS S3 bucket

- an IAM user or role with access to that bucket
- the installer kubeconfig and the values file from your environment's install page (`kamiwaza-values.yaml` in the examples)

The IAM identity should be allowed to perform at least:

- `s3:ListBucket`
- `s3:GetObject`
- `s3:PutObject`
- `s3:DeleteObject`

Choose S3 Before the First Install

Changing values does not migrate existing objects. On a running install, workroom files and Skills Library packages already in SeaweedFS are not copied to S3, and the values on this page set `seaweedfs.enabled: false`, which makes the next upgrade delete the SeaweedFS volume and every object on it. Skills Library packages stored there stay listed, but downloading them fails. To keep that data, leave `seaweedfs.enabled: true` until a separately planned and verified transfer has copied it. Moving from S3 back to SeaweedFS needs the same transfer.

Configuration Model

External S3 is configured in one block of your values file, under `core.context.objectStorage`:

```yaml
core:
  context:
    objectStorage:
      enabled: true
      defaultBucket: ""
      defaultRegion: ""
      defaultPrefix: "context/raw"
      endpointUrl: "https://s3.<region>.amazonaws.com"
      credentialsSecretRef:
        name: ""
        accessKeyIdKey: "access_key_id"
        secretAccessKeyKey: "secret_access_key"
        sessionTokenKey: "session_token"
```

- Set `endpointUrl` to your bucket's regional endpoint, `https://s3.<region>.amazonaws.com`, or to your S3-compatible service's endpoint. Do not leave it empty: on an install that ran with SeaweedFS, an empty value can leave core using the old SeaweedFS address.
- Set `enabled: true` explicitly. Without it, the ambient-credentials path (empty Secret name) does not turn object storage on.
- When this block is set, the chart uses it as written and ignores the SeaweedFS wiring. Setting `seaweedfs.enabled: false` alongside it removes the unused SeaweedFS deployment and volume.
- Setting `storage.workrooms.backend: s3` does not configure core. Use `core.context.objectStorage`.

This configuration is rendered into:

- the `core-config` ConfigMap for non-secret S3 settings
- the `core-scheduler`, `core-raycluster-head`, and `core-raycluster-worker` deployments for secret-backed AWS credentials

Option 1: Static AWS Access Keys

Use this option when your cluster does not already provide AWS credentials to the pods.

1. Create a Kubernetes Secret

Core pods wait on this Secret, so create it before you install or upgrade, in the `kamiwaza` namespace your platform administrator created in Platform Prerequisites (Kubernetes with Istio (see online documentation) or OpenShift with GreyMatter (see online documentation)). Build it from files, so the keys never appear in a command line or your shell history:

```
printf 'AWS secret access key: '; read -rs AWS_SECRET_ACCESS_KEY; echo
printf 'AWS session token (Enter if none): '; read -rs AWS_SESSION_TOKEN; echo
(umask 077; d=$(mktemp -d)
 printf '%s' '<aws-access-key-id>' > "$d/access_key_id"
 printf '%s' "$AWS_SECRET_ACCESS_KEY" > "$d/secret_access_key"
 [ -n "$AWS_SESSION_TOKEN" ] && printf '%s' "$AWS_SESSION_TOKEN" > "$d/session_token"
 kubectl -n kamiwaza create secret generic core-s3 --from-file="$d" \
   --dry-run=client -o yaml | kubectl apply -f -
 rm -rf "$d")
unset AWS_SECRET_ACCESS_KEY AWS_SESSION_TOKEN
```

The same command creates the Secret or replaces its keys. Core reads `session_token` when it is present; no values change is needed. Session credentials expire, so rotate the Secret before they do, as in Rotate credentials (see online documentation).

2. Add the S3 settings to your values file

Merge this into `kamiwaza-values.yaml`:

```yaml
core:
  context:
    objectStorage:
      enabled: true
      defaultBucket: "my-kamiwaza-artifacts"
      defaultRegion: "us-west-2"
      defaultPrefix: "context/raw"
      endpointUrl: "https://s3.us-west-2.amazonaws.com"
      credentialsSecretRef:
        name: "core-s3"
        accessKeyIdKey: "access_key_id"
        secretAccessKeyKey: "secret_access_key"

seaweedfs:
  enabled: false
```

Notes:

- `defaultBucket` is required; the render fails without it.
- If your Secret uses different key names, set `accessKeyIdKey` and `secretAccessKeyKey` to match.
- `endpointUrl` must match the bucket's region.
- This changes workroom storage only. Model storage keeps its configured backend.

3. Apply the values

Run the `helm upgrade --install` command from your environment's install page, for example Install on Kubernetes with Istio, step 3 (see online documentation), with the same chart version and the updated values file. The upgrade restarts the core deployments when their S3 settings change.

Option 2: Ambient AWS Credentials

Use this option if your cluster already provides AWS credentials to pods through IAM roles or another AWS-native mechanism.

In this case, do not create a Secret. Leave `credentialsSecretRef.name` empty, so no credential environment variables are rendered and the AWS SDK falls back to ambient credentials. `enabled: true` is **required** here: without it, the chart cannot tell that object storage is configured.

```
core:
  context:
    objectStorage:
      enabled: true
      defaultBucket: "my-kamiwaza-artifacts"
      defaultRegion: "us-west-2"
      defaultPrefix: "context/raw"
      endpointUrl: "https://s3.us-west-2.amazonaws.com"
      credentialsSecretRef:
        name: ""

seaweedfs:
  enabled: false
```

Apply the values as in Option 1, step 3.

Rotate credentials

To replace the keys, rerun Option 1, step 1 with the new values. The chart does not track the Secret's contents, so then restart the core deployments to make them read the new values. Confirm the Secret exists first; if it does not, the restarted core pods cannot start and the API is unavailable until it does:

```
kubectl -n kamiwaza get secret core-s3
```

```
kubectl -n kamiwaza rollout restart \
  deployment/core-scheduler deployment/core-raycluster-head deployment/core-raycluster-
worker
```

Verify the Configuration

Check the ConfigMap

Verify that the non-secret S3 settings are present:

```
kubectl -n kamiwaza get configmap core-config -o yaml | grep CONTEXT_SERVICE_S3
```

You should see values like:

```
CONTEXT_SERVICE_S3_DEFAULT_BUCKET: my-kamiwaza-artifacts
CONTEXT_SERVICE_S3_DEFAULT_PREFIX: context/raw
CONTEXT_SERVICE_S3_DEFAULT_REGION: us-west-2
CONTEXT_SERVICE_S3_ENABLED: "true"
CONTEXT_SERVICE_S3_ENDPOINT_URL: https://s3.us-west-2.amazonaws.com
```

Check the Credential Reference

If you are using Secret-backed credentials, verify the rendered credential reference without echoing any secret values:

```
kubectl -n kamiwaza get deploy core-scheduler -o yaml | grep -A4
CONTEXT_SERVICE_S3_ACCESS_KEY_ID
kubectl -n kamiwaza get deploy core-raycluster-head -o yaml | grep -A4
CONTEXT_SERVICE_S3_ACCESS_KEY_ID
```

The `secretKeyRef` should name your Secret (`core-s3`) and key (`access_key_id`).

Validate in the Product

Import a Skills Library package, then download it. Restart the core deployments as in Rotate credentials (see online documentation) and download it again; the package should still be there. Its objects appear in the bucket under the `defaultPrefix`.

Troubleshooting

Error: secret "core-s3" not found (CreateContainerConfigError)

Core pods (`core-scheduler`, `core-raycluster-head`, `core-raycluster-worker`) are stuck in `CreateContainerConfigError` (shown as `0/1`, or `1/2` on Istio-injected installs), the pod events show `secret "core-s3" not found`, and the install never completes: `helm upgrade --install` waits on post-install jobs while most other pods run normally.

This means core was configured to read S3 credentials from a Secret that does not exist in the `kamiwaza` namespace. It usually means a `core.context.objectStorage` block left in your values file from an earlier external-S3 setup is overriding the default SeaweedFS wiring.

Check which Secret core references:

```
kubectl -n kamiwaza get deploy core-scheduler -o yaml | grep -A4
CONTEXT_SERVICE_S3_ACCESS_KEY_ID
```

- `name: core-s3` — this page's external-S3 settings are in effect

- `name: seaweedfs-s3-credentials` — the default wiring is in effect

Fix:

- If you intended the default SeaweedFS storage: remove the `core.context.objectStorage` block from your values file, set `seaweedfs.enabled` back to `true` if you changed it, and rerun `helm upgrade --install`.
- If you intended external S3: create the Secret (Option 1, step 1). The kubelet retries automatically and the pods recover within a few minutes (container-creation backoff caps at 5 minutes). To retry immediately, restart the deployments as in Rotate credentials (see online documentation).

Error: Workroom storage is not configured for Skills Library

This usually means core does not see `CONTEXT_SERVICE_S3_DEFAULT_BUCKET`.

Check:

- the block is nested under `core:` — `core.context.objectStorage`, not top-level `context.objectStorage`. `helm get values kamiwaza -n kamiwaza` shows what Helm received.
- `enabled: true` is set
- the `core-config` ConfigMap contains `CONTEXT_SERVICE_S3_DEFAULT_BUCKET`

Error: Unable to locate credentials

This means the bucket settings are present but AWS authentication is not.

Check:

- the Secret exists in the `kamiwaza` namespace
- `credentialsSecretRef.name` matches the Secret name
- the Secret keys match the configured key names
- the core deployments were restarted after the Secret was created or changed

Error: AccessDenied or SignatureDoesNotMatch

This usually means the credentials are present but not valid for the target bucket or region.

Check:

- the bucket name
- the AWS region
- the IAM policy attached to the user or role
- whether the credentials belong to the AWS account that owns the bucket

Security Notes

- Do not store AWS secrets in workroom attributes or user-editable metadata.
- Prefer Kubernetes Secrets or ambient AWS identity over plain-text credentials in values files.

- Use an `https://` endpoint in `endpointUrl`.

Workroom Runtime Contract

Workrooms support runtime-scoped applications, shared membership, and a collaboration SSE feed. This page summarizes the public behaviors extension authors and operators can rely on when they integrate apps with workroom context.

Runtime app binding

A runtime application can be launched inside a workroom. While it is running in that workroom:

- its lifecycle (start, stop, restart) is scoped to the workroom
- its identity headers carry the workroom via `X-Workroom-Id`
- API calls it makes on behalf of a caller are evaluated against the caller's workroom membership and role, not the launcher's
- shared workroom credentials (connectors, model endpoints) are only exposed to the runtime when the caller is a member

When a user leaves a workroom, the platform refreshes any bound runtime sessions so the departing user's access is dropped. The refresh completes within the runtime launch token's TTL (minutes, not hours) - the outgoing user's token is not proactively revoked, but it is not renewed, so any in-flight call survives at most until the current token expires.

Membership and roles

The workrooms API exposes roster operations that runtimes and the built-in Workrooms page share (as does the deprecated Workroom Manager extension, on installations that still use it):

Endpoint	Purpose
`GET /api/workrooms/{workroom_id}/members`	List members visible to the caller.
`POST /api/workrooms/{workroom_id}/members`	Invite or add a member.
`PATCH /api/workrooms/{workroom_id}/members/{member_user_id}`	Update a member's role.
`DELETE /api/workrooms/{workroom_id}/members/{member_user_id}`	Remove a member (see active-session confirmation below).

Active-session confirmation mechanism. Confirmation is carried as the query parameter `?confirm_active_sessions=true` on the `DELETE` request:

1. Client issues `DELETE /api/workrooms/{workroom_id}/members/{member_user_id}` without the query parameter (or with `false`).

2. If the target has one or more active workroom sessions, the server returns **HTTP 409** with `detail: "Member has active workroom sessions; confirm removal to continue"`. The membership is **not** modified.
3. Client surfaces the confirmation prompt to the operator, then retries with `DELETE /api/workrooms/{workroom_id}/members/{member_user_id}?confirm_active_sessions=true`.
4. On the confirmed retry, the server removes the membership and terminates any bound runtime sessions for that user as part of the same request.

If the target has no active sessions, the initial `DELETE` succeeds without the query parameter. There is no request body or custom header - the mechanism is purely the query-string flag plus the `409` retry handshake.

Collaboration SSE feed

Clients can subscribe to a per-workroom SSE stream for presence and collaboration signals:

```
GET /api/workrooms/{workroom_id}/events/stream
Accept: text/event-stream
Authorization: Bearer <token>
```

Accepted bearer types. The `Authorization` header accepts any of the following, in order of preference:

- A **user JWT** issued by the deployment's IdP. Use this for interactive browser or SPA clients subscribed on behalf of a signed-in user. JWTs must be signed with `RS256`, include `exp`, and include `kid` so the deployment can resolve the signing key. Deployments validate `iss` and, when configured, `aud`.
- A **runtime launch token** issued for an extension runtime bound to this workroom. Use this when the subscriber is an extension runtime. The token's runtime + user + workroom tuple must match the `{workroom_id}` in the path.
- A **personal access token (PAT)** scoped to the caller. Use this for service-to-service or CLI subscribers that cannot complete an interactive IdP flow. The caller still must be a current workroom member.

Tokens that do not resolve to a current member of the workroom receive `403`. Expired tokens receive `401`; reconnect with a freshly issued token and the `Last-Event-ID` header for resumable delivery.

Event categories:

- **presence** - a member joins, leaves, or goes idle in the workroom
- **runtime** - a runtime app in the workroom changes state (starting, ready, stopping)
- **collaboration** - membership or role changes the caller has permission to see

Each event includes a stable `id`, an `event` type, and a JSON `data` payload. Clients should honor the `retry` hint and reconnect with the last event id for resumable delivery.

Rate limits and connection budget.

- One active SSE connection per `(user, workroom)` tuple is the supported contract. Opening a second connection with the same tuple may cause the platform to close the older connection; do not rely on fan-out across multiple tabs without a client-side shared worker.
- Server-side `retry` hints are advisory. On unexpected disconnects, reconnect with exponential backoff starting at 1s and capping at 30s, with jitter. Do not reconnect in a tight loop on `401` or `403` - re-authenticate first.

- The stream is not a durable queue. Events older than the server's in-memory buffer may be dropped; `Last-Event-ID` provides best-effort resumption, not guaranteed delivery. Clients that need an authoritative state should re-fetch the relevant REST resource after reconnect.

Behavior to rely on

- Workroom context isolation: a runtime cannot read data belonging to another workroom, even when the same user is a member of both.
- Credential gating: shared workroom credentials are distributed only to runtimes whose caller is a current workroom member.
- Fail-closed reads: ambiguous workroom reads are denied rather than silently broadened.

See Also

- Developer Guide - Workroom-scoped runtime sessions (see online documentation)

Administrator Guide

Use this guide to understand the security-related responsibilities of a Kamiwaza administrator in a customer deployment. It focuses on the current Kubernetes-based platform and avoids internal helper scripts, source-repo workflows, and host-local bootstrap procedures.

Authentication Model

Kamiwaza 1.3 runs with authentication enabled. It uses an external or bundled identity provider, validates tokens, and enforces policy. Running Kamiwaza with authentication turned off, sometimes called Lite mode, is not supported in 1.3.

For customer environments, use a customer-facing HTTPS domain, managed ingress, and centralized identity.

Identity Provider Responsibilities

Administrators are responsible for confirming that the identity provider integration is correct for the deployment.

At a minimum, verify:

- the deployment domain is correct
- redirect and callback URLs match the deployed hostname
- the audience or client ID expected by Kamiwaza is configured correctly
- issuer and JWKS metadata are reachable by the platform
- user claims required by policy are present

For more detail on tenant-aware authorization, see:

- Relationship-Based Access Control (ReBAC) (see online documentation)
- ReBAC Deployment Guide (see online documentation)

For CAC-enabled federal environments, also review:

- CAC Overview (see online documentation)

User and Role Management

User lifecycle management should be handled through the deployment's identity provider and normal administrative workflow.

Best practice:

- create administrator and standard-user accounts through the identity provider or supported admin workflow

- avoid relying on node-local or package-local CLI tools as the default customer user-management path
- review group and role mappings as part of onboarding
- disable or remove temporary test identities before production rollout

Kamiwaza commonly distinguishes between:

Role Type	Typical Purpose
administrator	Platform setup, access policy, troubleshooting, and environment validation
standard user	Daily use of models, apps, tools, and workrooms
service identity	Automation, extensions, or system-to-system integration

Secrets Management

Treat authentication, signing, and integration credentials as deployment-managed secrets.

Best practice:

- store secrets in Kubernetes Secrets or your approved external secret manager
- keep long-lived secrets out of plain-text values files whenever possible
- rotate administrative and client secrets through your normal release workflow
- avoid documenting or depending on file-system secret fallbacks inside running containers

This applies to credentials such as:

- identity provider client secrets
- administrative passwords
- signing keys and shared secrets
- object storage credentials

Consent Gate and Classification Banners

Kamiwaza can enforce a pre-login consent gate and show classification banners for the deployment.

Use these controls when your environment requires:

- an explicit user acknowledgment before access
- persistent classification or handling caveats across the UI
- consistent banner behavior for embedded experiences

For configuration details, see:

- Consent Gate and Classification Banners (see online documentation)

Relationship-Based Access Control

ReBAC adds tenant-aware and relationship-aware authorization on top of authentication.

Administrators should treat ReBAC as part of deployment configuration, not as an ad hoc runtime toggle.

Recommended responsibilities:

- confirm ReBAC is enabled only in auth-enabled environments
- validate tenant and relationship data before rollout
- verify representative allow and deny flows during release testing
- preserve evidence of policy and access validation for accreditation or audit needs

Use these docs together:

- ReBAC Overview (see online documentation)
- ReBAC Deployment Guide (see online documentation)
- ReBAC Validation Checklist (see online documentation)

Personal Access Tokens and Service Access

If your deployment uses automation, extensions, or system integrations, review how those workflows authenticate.

Best practice:

- issue only the minimum access required for the integration
- prefer managed service identities and approved token workflows
- rotate and revoke credentials when an integration changes ownership or scope
- keep tenant scoping aligned with the intended workload

Logging, Audit, and Troubleshooting

Security troubleshooting should use the environment's approved observability path.

That may include:

- the Kamiwaza UI log viewer
- Kubernetes-native logging
- OpenTelemetry or external logging systems configured for the deployment

Administrators should be able to verify:

- successful authentication events
- denied access events
- session expiration or revocation behavior

- enough request context to support incident response or accreditation evidence

Audit coverage

- All state-changing API operations emit an audit record (actor, target, action, request id, and outcome).
- A denied-request audit middleware records `401` and `403` responses so failed access attempts leave a trail even when the downstream handler is never invoked.
- Audit records are surfaced through the same observability path as other security logs.

What is not audited. Read-only `GET` requests that succeed are not individually audited — only denied reads (`401` / `403`) and state-changing operations are. Internal health probes, static-asset fetches, and identity-provider redirect callbacks are excluded to keep the audit trail signal-to-noise high. Payload bodies are not captured; audit records store identifiers and outcome, not request/response contents.

Retention and storage. Audit records are emitted to the deployment's configured observability sink (the same path as other security logs — Kubernetes-native logging, OpenTelemetry collector, or the Kamiwaza UI log viewer). Retention is governed by that sink's configuration, not by Kamiwaza itself: set retention on your log backend (for example, Loki, Elasticsearch, or a SIEM) to match your accreditation requirements. There is no Kamiwaza-internal audit database with its own retention window.

Authenticated-by-default API surface

The platform now applies the `AuthenticatedUser` dependency to all non-exempt endpoints, including admin and destructive routes. The effective contract:

- Every API endpoint requires an authenticated caller unless explicitly exempt (health, login, public metadata).
- Admin and destructive endpoints additionally require the appropriate role or scope.
- The OpenAPI spec now documents auth requirements per endpoint — use it as the source of truth when integrating. The spec is served by the platform at `/api/openapi.json` (with an interactive browser at `/api/docs`) on the deployment's own hostname.

Cluster trust and machine-to-machine federation

For multi-cluster and federated deployments, identity headers from peer services are only honored when signed by a trusted cluster identity. Unsigned requester-identity headers are now distrusted and dropped. Operators federating two Kamiwaza clusters must exchange cluster trust material as part of deployment; contact Kamiwaza Support for the per-environment procedure.

Rotation, revocation, and compromise. Cluster trust material is long-lived but not permanent. Rotation, revocation on decommission, and response to suspected key compromise are deployment-specific procedures and are not exposed as a self-service UI in the platform. If you need to rotate or revoke a peer cluster's trust material, or if you suspect a cluster identity has been compromised, contact Kamiwaza Support for the per-environment procedure — do not attempt to edit signing material in running pods. Until a compromised identity is revoked through the supported path, isolate the affected cluster at the network layer.

Local (non-SSO) user password policy

When local authentication is enabled, new users must supply a valid email address and a password meeting the platform's strength policy at creation. Administrators provisioning local users should budget for:

- distributing per-user email addresses (no shared logins)
- communicating the password-strength requirements to end users
- rotating any seeded credentials that pre-date the policy

Policy specifics.

- **Minimum length:** 12 characters. Shorter passwords are rejected at creation and at rotation.
- **Complexity:** the platform does not enforce a fixed character-class rule (upper/lower/digit/symbol); length is the primary control. Deployments that require explicit complexity rules should federate to an external IdP that enforces them.
- **Reuse and expiry:** the local store does not enforce password history or forced expiry. If these are required for accreditation, use an external IdP (for example, Keycloak) rather than local auth.
- **Lockout:** the local store does not implement per-user lockout on repeated failed logins. Rate-limiting and lockout should be enforced at the IdP or at the ingress layer for deployments that require it.
- **Hash at rest:** passwords are stored as PBKDF2-HMAC-SHA256 with a per-user random salt and 200,000 iterations. Plaintext passwords are never persisted. The hash and salt are held in the platform's identity store alongside the rest of the user record.

For environments with stricter requirements (rotation cadence, history depth, lockout, complexity classes), run in auth-enabled mode against your IdP rather than relying on local auth.

For broader deployment logging guidance, see:

- Observability (see online documentation)

Ephemeral Sessions for App Garden

Some deployments use ephemeral app sessions so launched applications are automatically cleaned up when the user logs out or the session expires.

Use this behavior when you want:

- reduced persistence for temporary app launches
- automatic cleanup for demos or evaluation environments
- stronger alignment between user session state and app lifetime

If your deployment enforces ephemeral extension behavior globally, document that choice as part of rollout readiness so administrators and users understand the lifecycle impact.

Administrator Validation Checklist

Before handing the environment to users, confirm:

- users can sign in at the expected HTTPS hostname
- role mappings behave as expected
- required banners or consent messaging appear correctly
- protected actions allow the right users and deny the wrong ones
- logs and audit evidence are visible through the approved operations path
- secrets and callback URLs are managed through the deployment workflow, not manual pod edits

Next Steps

- Use Quickstart (see online documentation) for post-install validation.
- Use Configuration Reference (see online documentation) for deployment configuration patterns.
- Use CAC Overview (see online documentation) for federal CAC-specific guidance.

Consent Gate and Classification Banners

Kamiwaza can enforce a pre-login consent gate and display classification banners across the UI and any embedded apps. This is designed for security and compliance programs that require a user acknowledgment before access, plus persistent system-high markings.

What this provides

- **Consent gate**: A modal overlay shown before login that requires explicit acceptance.
- **Classification banners**: Top and bottom banners with configurable text and colors (for example "SECRET").
- **Embeddable script**: A single JavaScript include that applies the same behavior in external apps.

Public endpoints

These endpoints are intentionally public (no auth required) so they can be used before login.

- `GET /api/security/public/config`
  - Returns the consent/banner configuration consumed by the UI and embed script.
- `POST /api/security/consent/accept`
  - Records consent acceptance for audit purposes (client IP + user agent).
- `GET /api/security/embed.js`
  - Returns the embeddable JavaScript bundle for banners and consent gate.

Embedding the script

Add the following tag to any app that should mirror Kamiwaza's consent and banner behavior:

```
<script src="https://<gateway-host>/api/security/embed.js"></script>
```

The script:

- Fetches `/api/security/public/config`
- Renders banners at the top and bottom of the page
- Enforces a consent gate until accepted
- Fails closed (shows the gate with a retry option if config fetch fails)

Consent is tracked in session storage for the browser session and is also recorded server-side for audit logs.

Configuration

Deployment settings

Set these through your deployment values, ConfigMaps, or equivalent environment configuration workflow:

Variable	Description	Default
`KAMIWAZA_SECURITY_CONSENT_ENABLED`	Enable the consent gate	`false`
`KAMIWAZA_SECURITY_CONSENT_BUTTON_LABEL`	Custom button label	`Accept`
`KAMIWAZA_SECURITY_BANNER_ENABLED`	Enable classification banners	`false`
`KAMIWAZA_SECURITY_BANNER_TOP_TEXT`	Text for the top banner	(none)
`KAMIWAZA_SECURITY_BANNER_TOP_COLOR`	Hex color for top banner	(none)
`KAMIWAZA_SECURITY_BANNER_BOTTOM_TEXT`	Text for the bottom banner	Defaults to top text
`KAMIWAZA_SECURITY_BANNER_BOTTOM_COLOR`	Hex color for bottom banner	Defaults to top color

Consent content file

Consent HTML is loaded from:

```
$KAMIWAZA_ROOT/config/security/consent.html
```

If the file is missing, a default short message is used. In customer deployments, manage any custom consent content through the supported configuration and release process for your environment rather than editing files directly in running containers.

Operational notes

- Consent and banners are purely UI-level controls. They do not replace API authentication or authorization.
- For system-high or CAPCO-aligned deployments, keep banner text aligned with your site classification policy.
- If you embed the script in external apps, make sure those apps are reachable through the same gateway so the script can resolve the correct API origin.

Relationship-Based Access Control Overview

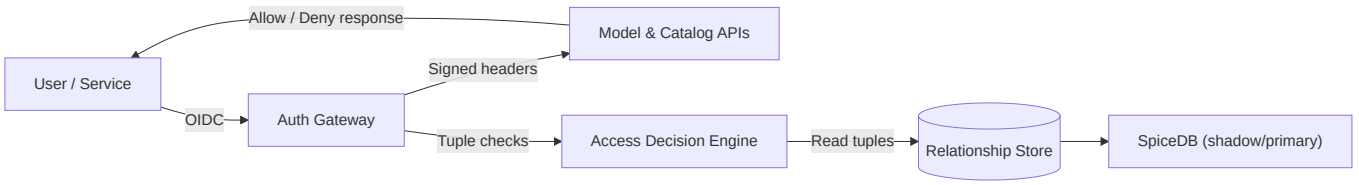
> **UNRELEASED DEFAULT CHANGE**
>
> The first release that makes SpiceDB the default ReBAC backend uses a maintenance window with a service outage for installation, upgrade, and planned graph-authoritative API replica starts. Its version has not yet been assigned here. The shadow cutover guidance below does not replace the SpiceDB maintenance upgrade procedure (see online documentation) for that release.

Kamiwaza uses relationship-based access control (ReBAC) to enforce tenant-scoped and role-aware policies across the platform. This page summarizes the capabilities delivered with the current release, how to integrate an identity provider, and the operational practices required to keep the system healthy.

What ReBAC Delivers

- **Deny by default** – every protected API checks for an explicit `subject → relation → resource` tuple before permitting access.
- **Context-aware policies** – relations cover ownership, editor/viewer roles, clearance tiers, and nested group membership.
- **Runtime updates** – policy manifests and tenant bootstrap files can be replayed without restarting services.
- **Auditable decisions** – each authorization call records the decision, reason, backend, and correlation ID.
- **Shadow cutover** – Postgres and SpiceDB can run in parallel to verify parity before promoting SpiceDB to the primary backend.
- **Automatic ownership tuples** – when `AUTH_REBAC_ENABLED=true`, catalog/model uploads seed owner/editor/viewer tuples automatically so creators can immediately list and download their assets.

Architecture Snapshot



Component	Purpose	Notes
Auth Gateway	Terminates OIDC sessions, issues personal access tokens (PAT), forwards signed headers.	Keycloak is the reference IdP; OIDC, SAML, and CAC flows are supported when configured.
Access Decision Engine	Evaluates tuples stored in Postgres or SpiceDB and returns permit/deny with reasoning.	Targets <100 ms latency for typical requests.
Relationship Store	Persists tuple data and mirrors to SpiceDB when enabled.	Ships with default tuples for system tenants.
Service Guards	Wrap API endpoints with `enforce_*` helpers to require specific relations.	Catalog dataset deletes and model deletes ship protected in this release.
Policy Assets	`configs/rebac/policies/*.yaml` and tenant bootstraps under `configs/rebac/tenants/`.	Validate before applying changes.

Identity Provider Integration

1. **Create a confidential client** in your identity provider (Keycloak in the reference deployment).
   - Client ID must match `AUTH_GATEWAY_JWT_AUDIENCE` (default `kamiwaza-platform`).
   - Allowed redirect URI: the external URL of the Kamiwaza deployment (`https://<gateway>/api/auth/callback`).
2. **Expose required claims** in the ID token or user info endpoint.
   - Include `roles`, `tenant`, and any clearance attributes referenced in policies.
   - Ensure the token carries `exp`, `iat`, `sub`, and `iss` claims; Kamiwaza validates them on every request.
3. **Configure the Auth Gateway** through your deployment values, Kubernetes ConfigMaps, and Secrets.
   - Set the issuer and JWKS URL to match your Keycloak realm (`https://<keycloak>/realms/<realm>` and `/protocol/openid-connect/certs`).
   - Align `AUTH_GATEWAY_JWT_AUDIENCE` with the confidential client you created (`kamiwaza-platform` by default).
   - Enable ReBAC (`AUTH_REBAC_ENABLED=true`) and choose your primary backend (`postgres` or `spicedb`).
   - Point the session store at your Redis deployment (`AUTH_REBAC_SESSION_REDIS_URL=rediss://<redis-host>:6380/0`).
   - Define the default tenant fallback (`AUTH_REBAC_DEFAULT_TENANT_ID="__default__"` for single-tenant labs) and enable PAT tagging (`AUTH_PAT_TENANT_TAGGING_ENABLED=true`) so newly issued tokens include tenant metadata.
   - See the ReBAC Deployment Guide (see online documentation) for the customer-facing rollout sequence and validation guidance.
4. **Optional PAT workflow** – use the Auth gateway PAT endpoint to issue automation tokens that embed tenant metadata.

Policy Management

1. **Review your deployment's policy and tenant artifacts** to understand the default relations for models, datasets, and administrative roles.
2. **Validate changes before rollout** using the supported administrative or deployment workflow for your environment.
3. **Promote tenant and policy data through change management** rather than editing live pods directly.
4. **Capture evidence** such as policy revisions, release records, and validation results for auditability.

Operations Checklist

- **Gateway policy** – enable `default_deny: true` and explicitly list the `/api/*` paths users require.
- **Shadow validation** – run with `AUTH_REBAC_BACKEND=postgres` and `AUTH_REBAC_SHADOW_COMPARE=true` until SpiceDB decisions match Postgres, then cut over.
- **Monitoring** – tail structured `rebac_decision` logs today; metrics endpoints will be documented when generally available.
- **Session store** – ensure Redis is served over TLS (`rediss://`), authentication is enabled, and session TTLs match your security policy (default 8 hours).
- **Audit artifacts** – capture policy validation output, tuple plans, and decision logs for the release record.

Demonstrating ReBAC

For a walkthrough that exercises authentication, tuple enforcement, and observability expectations, follow the ReBAC Validation Checklist (see online documentation). It covers customer-facing sign-in, allow/deny checks, and log verification.

Limitations & Roadmap

- ReBAC guards cover catalog, models, DDE connectors/documents, and retrieval job access. Additional services will continue to adopt guard coverage over time.
- Some ingestion workflows are still admin-scoped; expand tuple policies if you need finer-grained controls for your administrators.

Need help integrating ReBAC or exporting evidence for accreditation? Reach out to your Kamiwaza support representative for the latest runbooks and automation scripts.

Next steps

- Follow the ReBAC Deployment Guide (see online documentation) to configure the deployment.
- Run the ReBAC Validation Checklist (see online documentation) to validate tuple enforcement and logging.

ReBAC Deployment Guide

Use this guide to enable relationship-based access control (ReBAC) in a customer deployment of Kamiwaza. It is written for the current Kubernetes-based platform and assumes configuration is managed through your deployment values, Kubernetes Secrets, and cluster release workflow.

This page intentionally does not cover internal helper scripts, source-repo bootstrap commands, or pod-local file edits.

> **UNRELEASED DEFAULT CHANGE**
>
> The first release that makes SpiceDB the default ReBAC backend requires a maintenance window with a service outage for installation, upgrade, and planned graph-authoritative API replica starts. Its version has not yet been assigned here. Follow the SpiceDB maintenance upgrade procedure (see online documentation) for preparation, timing, and validation. The PostgreSQL example and rollout sequence below describe the earlier configuration; they do not replace that release's maintenance procedure or imply a PostgreSQL fallback during an outage.

Before You Start

Confirm the deployment meets these prerequisites:

- Kamiwaza is deployed on Kubernetes and reachable at its customer-facing HTTPS hostname.
- Authentication is enabled for the environment.
- You have administrator access to your identity provider.
- You have a supported release workflow for updating Helm values, Kubernetes Secrets, or the equivalent deployment inputs.
- You have a plan for tenant bootstrap data and policy validation before rollout.

For general installation and environment preparation, see:

- System Requirements (see online documentation)
- Installing Kamiwaza (see online documentation)
- Configuration Reference (see online documentation)

ReBAC Deployment Model

In the current platform, ReBAC is enabled as part of the auth-enabled Kubernetes deployment.

At a high level:

1. Configure the customer-facing domain and authentication settings.
2. Enable ReBAC in your deployment values.
3. Configure the identity provider so tokens contain the claims Kamiwaza expects.

4. Provide tenant bootstrap and policy data through your administrative release workflow.
5. Validate sign-in, allow/deny behavior, and audit logging before production use.

Core Deployment Settings

The exact value names may vary slightly by deployment package, but the current reference deployment uses settings equivalent to these:

```
global:
  domain: kamiwaza.example.com

auth:
  enabled: true

rebac:
  enabled: true
  backend: postgres
  allowCommunityFallback: false
```

Key points:

- `global.domain` should match the canonical customer-facing HTTPS hostname.
- `auth.enabled` must be on for ReBAC-enabled deployments.
- `rebac.enabled` turns on tuple-based authorization checks.
- `rebac.backend` should match the supported backend for your environment.
- `rebac.allowCommunityFallback` should remain disabled outside of narrowly scoped lab scenarios.

Do not manage these settings by editing pod-local files or source-controlled `env.sh` files in a running environment.

Identity Provider Configuration

Kamiwaza expects a standards-based OpenID Connect deployment. Keycloak is the reference identity provider, but the same requirements apply to equivalent OIDC providers.

Configure the identity provider as follows:

1. Create a confidential client for Kamiwaza.
2. Set the redirect URI to:
   - `https://<your-domain>/api/auth/callback`
3. Set the audience/client ID expected by Kamiwaza.
4. Ensure the issuer and JWKS endpoints are reachable by the platform.
5. Include the claims your policies depend on, such as:
   - subject identifier
   - role membership

    - tenant identifier
    - clearance or attribute claims, if your policies require them

If you are deploying a CAC-enabled federal environment, also review:

- CAC Overview (see online documentation)

Session Store and Secrets

Production ReBAC deployments should use managed secrets and a secured session store.

Best practice:

- store client secrets, signing material, and administrative credentials in Kubernetes Secrets or your external secret manager
- keep session storage on a managed Redis deployment with TLS enabled where required by policy
- avoid inline plaintext credentials in values files

For broader auth and secrets guidance, see the Administrator Guide (see online documentation).

Policy and Tenant Data

ReBAC depends on policy definitions and tenant relationship data. These artifacts should be treated like configuration, not like ad hoc runtime edits.

Best practice:

- validate policy and tenant manifests before applying them
- promote them through the same change-management path you use for other customer configuration
- keep a record of the policy version or manifest revision used for each environment
- avoid editing authorization data directly inside running pods

If your environment is managed by Kamiwaza or by an internal platform team, use the supported administrative workflow for loading or updating tenant bootstrap data.

Recommended Rollout Sequence

Use this order for a new ReBAC-enabled deployment:

1. Configure the domain, ingress, and TLS.
2. Enable authentication and confirm browser sign-in works.
3. Enable ReBAC in deployment values.
4. Apply the tenant and policy artifacts for the target environment.
5. Validate role-based access with representative administrator and standard-user accounts.
6. Review logs and audit evidence before broad rollout.

What To Validate

Before marking the deployment ready, confirm:

- users can sign in through the configured identity provider
- session validation succeeds after login
- an authorized user can access protected resources
- an unauthorized user receives a deny response for the same resource class
- decision and auth logs are visible through the deployment's approved logging path

Use the ReBAC Validation Checklist (see online documentation) as the final sign-off path.

Troubleshooting Guidance

If validation fails, check these areas first:

- mismatch between the public domain and the configured callback URL
- missing or incorrect token claims from the identity provider
- missing tenant bootstrap or relationship data
- incorrect audience, issuer, or JWKS configuration
- session store connectivity or secret mismatches

For broader auth troubleshooting, see:

- Administrator Guide (see online documentation)
- Observability (see online documentation)

Next Steps

- Review the ReBAC Overview (see online documentation) for architecture and capability context.
- Complete the ReBAC Validation Checklist (see online documentation).
- Keep policy, tenant, and secret changes in your normal deployment release workflow.

ReBAC Validation Checklist

Use this checklist after enabling authentication and ReBAC in a customer environment. It is designed for public, customer-facing deployments and avoids internal bootstrap helpers, seeded demo users, and local compose workflows.

Before You Begin

Confirm these prerequisites:

- the environment is reachable at its customer-facing HTTPS hostname
- authentication is enabled and users can reach the sign-in flow
- ReBAC is enabled in the deployment
- at least two test accounts exist in the identity provider:
  - one account that should be allowed to manage or view the target resource
  - one account that should be denied for the same protected action
- you have access to the approved log or observability path for the environment

If you still need to configure the environment, start with the ReBAC Deployment Guide (see online documentation).

1. Browser Sign-In

1. Open the Kamiwaza web application.
2. Sign in with a known-good administrator or authorized user account.
3. Confirm you are returned to the application without an auth error or redirect loop.

Expected result:

- sign-in completes successfully
- the UI loads normally

2. Session Validation

Confirm that the environment recognizes an authenticated session.

One common check is:

```
curl -i https://<your-domain>/api/auth/validate
```

Run this with the session or bearer-token method approved for your environment.

Expected result:

- HTTP `200`
- authenticated user context is returned

3. Allow Path

Using an account that should have access:

1. open a representative protected workflow, such as a model, dataset, or other managed resource
2. confirm read access succeeds
3. if the role is expected to write or administer the resource, confirm one representative change also succeeds

Expected result:

- the permitted action completes successfully

4. Deny Path

Using an account that should not have access to that same action:

1. attempt the same protected operation
2. confirm the request is denied

Expected result:

- the user receives a deny response or equivalent UI error
- the request does not silently succeed

5. Tenant and Role Scope

If the environment is multi-tenant or uses tenant-scoped policy:

1. sign in as a user from the intended tenant
2. confirm that tenant-scoped resources are visible as expected
3. sign in as a user from a different tenant or role scope
4. confirm those resources are not exposed outside the allowed scope

Expected result:

- users see only the resources and actions granted to their tenant and role context

6. Logging and Auditability

Review the environment's approved logging path, such as:

- the Kamiwaza UI log viewer
- platform observability dashboards
- Kubernetes logs collected by your normal operations tooling

Confirm you can find records associated with:

- successful authentication
- denied access decisions
- the relevant correlation or request identifiers, if your environment exposes them

Expected result:

- logs are available for both allow and deny scenarios
- security and operations teams can trace the request outcome

7. Session Controls

If the deployment uses session revocation, inactivity timeout, or ephemeral-session behavior:

1. verify the expected timeout or logout behavior with a test account
2. confirm the user must re-authenticate after the session expires or is revoked

Expected result:

- session policy behaves as configured for the environment

8. Federal or CAC Validation

For CAC-enabled federal deployments, also validate:

- the client certificate is accepted through the ingress path
- the mapped user is correctly identified by the identity provider
- certificate-related failures produce clear deny behavior

Use:

- CAC Overview (see online documentation)

Success Criteria

Mark the environment validated when all of the following are true:

- sign-in works for intended users
- authorized actions succeed
- unauthorized actions are denied
- tenant boundaries behave as expected
- security-relevant logs are visible through the approved operations path

Capture the evidence your organization requires, such as screenshots, log excerpts, or ticket references, as part of deployment sign-off.

Cluster Federation

Kamiwaza cluster federation enables cross-cluster operations between paired Kamiwaza clusters. A federated cluster can query remote data, list remote models, and execute compute jobs on its partner — all through a secured mesh proxy. The transport is HMAC-signed and mTLS-protected; **who the caller is** is decided by the federation's identity mode (see online documentation), and **what they may do** is enforced by the receiving cluster's own ReBAC and per-record gates.

Key Concepts

Mesh Proxy — Routes API calls from one cluster to another. For transport integrity the proxy signs each request with the federation's pre-shared key (PSK) via HMAC-SHA256; it forwards the caller's own token so the receiver can validate the caller's identity, and routes to the remote cluster's Istio ingress gateway. The response streams back transparently.

Federation Trust — Each federation pair shares a pre-shared key. All cross-cluster requests are HMAC-signed with this key, and the receiving cluster verifies the signature before processing the request. The PSK/HMAC secures the *transport*; the caller's *identity* is validated separately, according to the federation's identity mode (see online documentation). Trust is per-federation, not per-cluster.

Identity Modes — A federation's identity mode (see online documentation) decides whose identity a cross-cluster caller presents and how the receiver validates it. Kamiwaza 1.3.4 supports `shared_idp` (receiver-controlled shared realm) and `peer_kc` (source-trusted compatibility mode, gated by cluster policy). `receiver_realm` is reserved for a future receiver-owned guest-identity workflow and is rejected in 1.3.4.

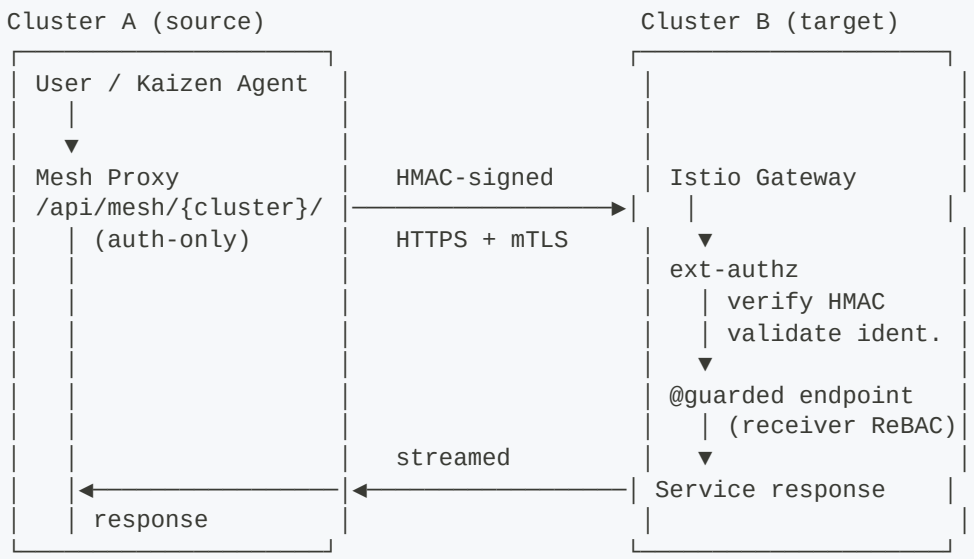
Receiver-controlled authorization — Cross-cluster egress on the source is **authenticated-only**: any authenticated caller may reach the mesh proxy — there is no `operator`-relation gate. Authorization is decided by the **receiving** cluster, which validates the caller's identity per the identity mode and then evaluates the request against its **own** per-resource ReBAC guards (catalog, retrieval, jobs) and per-record gates. Roles do **not** cross clusters (see shared identity ≠ shared authority (see online documentation)).

Compute Ships to Data — Raw data never leaves the originating cluster. Only structured results (API responses, job outputs) cross the federation boundary.

What You Can Do

Operation	How	What Crosses the Network
Query remote catalog	`GET /api/mesh/{cluster}/api/catalog/datasets/`	Dataset metadata (names, schemas, URNs)
Retrieve remote data	`POST /api/mesh/{cluster}/api/retrieval/jobs`	Query results (rows matching your filter)
List remote models	`GET /api/mesh/{cluster}/api/serving/deployments`	Deployment metadata (model names, status)
Run remote inference	`POST /api/mesh/{cluster}/runtime/models/{id}/v1/chat/completions`	Inference results
Submit remote job	`POST /api/mesh/{cluster}/cluster/jobs/run`	Structured job result (MVP+1)

Architecture



Prerequisites

- Two compatible Kamiwaza 1.3.4 clusters, each installed with Install on Kubernetes with Istio (see online documentation)
- STRICT mTLS (`PeerAuthentication`) in both namespaces
- Network connectivity between clusters on port 443
- Distinct, mutually reachable peer hostnames covered by each gateway TLS certificate. When connecting by IP, set `remote_ips[].hostname` for Host/SNI routing. See Federation setup (see online documentation) for the full list.
- A receiver-controlled identity mode. Use `shared_idp` for the first-wave workflow documented here; keep `peer_kc` default-off unless its weaker trust posture is explicitly accepted.

Federation lifecycle at a glance

Step	What	Where it is documented
Pair two clusters	Preflight both routes, create reciprocal records, and let the handshake exchange CA trust	Setup (see online documentation)
Allowlist federated users	Decide which remote users can act on this cluster	Setup (see online documentation)
Query remote data	Federated retrieval over the mesh proxy	Retrieval (see online documentation)
Submit remote jobs	Cluster-job execution with on-behalf-of (OBO) identity	Job Submission (see online documentation)
Apply policy gates	Cluster `ExecutionGate` and per-dataset `AttributeGate` bindings	Execution Gates (see online documentation)
Install custom gates	Ship custom policy code as Python packages	Gate Packages (see online documentation)
Day-2 operations	Diagnose, cancel, revoke, audit	Operations (see online documentation)

Next Steps

- Federation Setup Guide (see online documentation) — Create and pair federations
- Federated Retrieval (see online documentation) — Query remote data through the mesh proxy
- Job Submission (see online documentation) — Submit federated jobs with OBO identity
- Execution Gates (see online documentation) — Cluster `ExecutionGate` + dataset `AttributeGate` policy bindings

- Gate Packages (see online documentation) — Install custom gate classes as hash-pinned Python packages
- Operations & Troubleshooting (see online documentation) — Day-2 federation operations
- API Reference (see online documentation) — Endpoint catalog

Identity Trust Modes

When you pair two clusters, you decide **whose identity a cross-cluster caller presents to the remote cluster, and how the remote cluster verifies it.** This is the federation's **identity mode** — a per-federation setting chosen by the cluster that receives the calls (the *grantor*).

The current release and development baselines support two identity modes:

Mode	Trust	Who validates the caller	Use when
`shared_idp`	Receiver-controlled	The receiver validates the caller's token against a **shared realm** both clusters trust	Both clusters can trust one shared identity provider (single operator, or a tightly-coupled pair)
`peer_kc`	Source-trusted (legacy 1.0)	The receiver validates against the **peer cluster's own** Keycloak realm	Grandfathered pairings, or when you explicitly accept the source cluster as an identity authority

Core principle: shared identity ≠ shared authority

Trusting a shared identity does **not** grant shared authority. When a caller from another cluster presents a validated token, that token confers **identity only**:

- **Roles do not cross clusters.** A shared-realm caller's cluster-admin role is stripped; what they can do on the receiver is governed entirely by the **receiver's own** relationship-based access control (ReBAC).
- **Attributes come from the validated token, never from forwarded headers.** Attributes such as `clearance` or `compartment` are read from the caller's own signature-verified token. The receiver **ignores** any attribute headers the source cluster forwards.
- **Attributes are read from the token the caller presents**, not fetched live from the identity provider. A downgrade (for example, lowering a `clearance`) takes effect on the caller's **next** token, bounded by the token's lifetime.

This is why `shared_idp` is *receiver-controlled*: the receiver decides who its users are (via the shared realm it trusts) **and** what they may do (via its own ReBAC and per-record gates).

`shared_idp` — receiver-controlled

Both clusters trust the **same** realm as an identity provider. A caller authenticates against that shared realm and presents its token; the receiver validates that token against the realm's published keys (JWKS) before honoring the request.

Per-federation configuration (set on the receiver at pairing time):

Field	Purpose
`shared_issuer_url`	The shared realm's issuer URL. **Supplying this at create time selects `shared_idp`.**
`shared_jwks_url`	Ignored. The receiver fetches the signing keys from the URL derived from `shared_issuer_url` (`<issuer>/protocol/openid-connect/certs`), so a federation can't pair a trusted issuer with other keys. The field is still accepted.
`shared_ca_pem`	(Optional) TLS trust root used to reach the shared realm's JWKS endpoint.

> **IMPORTANT**
>
> Naming an issuer on a federation is **not sufficient** for the receiver to accept its tokens. The receiver only accepts a shared-realm token if that issuer is **enrolled in the cluster's trusted-shared-issuers list** (see Cluster policy (see online documentation) below). An unenrolled issuer is rejected.

`peer_kc` — source-trusted (legacy)

The receiver validates the caller against the **peer cluster's own** Keycloak realm — it trusts the source cluster to be the identity authority. This is the 1.0 model.

- Creating a **new** `peer_kc` federation is **refused** unless the cluster policy `ALLOW_UNTRUSTED_FEDERATION` is enabled (it is **off by default**). See Cluster policy (see online documentation).
- A new `peer_kc` federation is created in **strict mode** (`require_peer_jwt`): the caller's peer token must validate against the pinned peer realm.
- A federation created before 1.1 has **no identity mode set** — it is treated as a **grandfathered** `peer_kc` pairing and keeps working after upgrade, exempt from the `ALLOW_UNTRUSTED_FEDERATION` refusal, and surfaced as the weaker posture so operators can see which pairings rest on trusting the source.

`peer_kc` remains supported. It is not the recommended mode for a new pair, but it is not scheduled for removal. Keep it disabled unless the operator explicitly accepts the source-trusted posture.

Future: `receiver_realm` — receiver-owned guests

The planned `receiver_realm` mode provisions and owns a separate guest identity for every approved remote user. Pairing will use a receiver-owned request/approve flow, and each user will complete an onboarding request that the receiver approves before issuing guest credentials.

This mode is **not available in the current release or `develop` baseline**. Core does not expose receiver-realm creation, request/approve, guest, or onboarding routes, and mesh ingress rejects a stored `receiver_realm` value with `identity_mode_unsupported`. The mode remains planned for the Federation Keycloak Patterns work; constants, schema fields, SDK helpers, and gated tests do not make it a deployable capability.

The `shared_idp` setup in this guide does not exercise `receiver_realm`. Do not infer receiver-realm behavior from a shared-IDP pairing.

Choosing and configuring a mode

The **grantor** (the cluster receiving the calls) decides the mode from its own policy at pairing time. A requester may *propose* a mode, but an inbound proposal is not auto-trusted — the receiver's policy decides.

The mode is selected implicitly by what you supply when you create the federation:

- **Supply** `shared_issuer_url` → the federation is `shared_idp`.
- **Omit it on the legacy create path** → the federation is `peer_kc` (subject to `ALLOW_UNTRUSTED_FEDERATION`).
- **Do not select** `receiver_realm` → the reserved value is rejected until the receiver-owned guest-identity implementation is delivered and qualified.

> **WARNING**
>
> **The identity mode cannot be changed in place.** Switching a federation between `peer_kc` and `shared_idp` requires **deleting and re-pairing** it. (Rotatable trust details, such as `shared_ca_pem`, *can* be updated without re-pairing.)

See the Setup Guide (see online documentation) for the concrete pairing steps.

Cluster policy

Two cluster-level settings govern identity trust across **all** of a cluster's federations. Both are operator-configured (via the deploy chart) and default to the secure posture.

Setting	Deploy chart key	Default	Effe
`ALLOW_UNTRUSTED_FEDERATION`	`scheduler.allowUntrustedFederation`	`false`	Whe `fal` crea **new** `pee` (sou trust fede refus Exist pairi gran
`AUTH_GATEWAY_TRUSTED_SHARED_ISSUERS`	`scheduler.trustedSharedIssuers`	*(empty)*	The sepa of sh realr URL clust acce `sha` toke **Emp no s issu trust** `sha` toke rejec unles issue this

Next steps

- Setup Guide (see online documentation) — pair clusters and choose a mode
- Federated Retrieval (see online documentation) — cross-cluster retrieval with per-record gating
- Overview (see online documentation) — how federation fits together

Federation setup

This guide pairs two Kamiwaza clusters using the receiver-controlled `shared_idp` identity mode. The receiver validates the caller's shared-realm token, then applies its own brokered-user allowlist, ReBAC, and policy gates.

It applies to Kamiwaza 1.3.4 clusters.

`receiver_realm` is reserved for the future receiver-owned guest-identity workflow and is rejected by Kamiwaza 1.3.4. `peer_kc` remains available for compatibility, but new `peer_kc` pairs are refused unless the operator explicitly enables `ALLOW_UNTRUSTED_FEDERATION`.

Prerequisites

- Two Kamiwaza 1.3.4 clusters, each installed with Install on Kubernetes with Istio (see online documentation) and each serving its own domain. This guide uses `fed-a.example.com` and `fed-b.example.com`; substitute your own. Keep each cluster's values file; step 0 adds to it.
- HTTPS reachability in both directions: each cluster reaches the other's ingress gateway on port 443.
- Hostname peer addresses, or an ingress gateway that preserves the caller's source address. See Use hostnames for the peer address (see online documentation).
- Peer name resolution inside each cluster. See Resolve the peer's domain (see online documentation).
- Edge certificates each cluster trusts. See Trust the peer's certificate (see online documentation).
- One independently owned shared OIDC realm, its issuer URL, and a way for shared-realm users to obtain access tokens from it. Both clusters must reach the issuer over HTTPS and trust its certificate. The receiver fetches the realm's signing keys from `<issuer>/protocol/openid-connect/certs`, the Keycloak path, so the issuer must serve them there.
- The `admin` user on each cluster.

Resolve the peer's domain

Kamiwaza calls the peer and the shared realm's issuer from inside the cluster, so pods on each cluster must resolve the other cluster's domain to an address that reaches the peer's gateway, and the issuer's host to the issuer. If your DNS already answers for both domains and pods use it, nothing more is needed. Otherwise add a cluster DNS entry on each cluster. With CoreDNS, add a `hosts` block to the Corefile, above the `kubernetes` plugin. On `fed-a`:

```
hosts {
  10.0.0.12 fed-b.example.com
  10.0.0.20 idp.example.com
  fallthrough
}
```

`10.0.0.12` is an address of `fed-b`'s ingress gateway that `fed-a`'s pods can reach, and `10.0.0.20` is the issuer's. Make the reciprocal entry, and the issuer entry, on `fed-b`. Distributions that manage the CoreDNS configuration, k0s among them, need the change made through their own configuration, as described in DNS (see online documentation). Allow a minute or two for CoreDNS to reload before testing.

Use hostnames for the peer address

Give each `remote_ips` entry the peer's domain as its `ip`, for example `{"ip": "fed-a.example.com", "primary": true}`. The peer's domain must resolve inside the cluster; see Resolve the peer's domain (see online documentation).

A receiver accepts a pairing request only from its peer. For a hostname entry, the request's signature is that check, so the pre-shared key is the only proof that a request comes from the peer: handle it like a password. For a literal IP entry, the receiver also requires the request to arrive from that address, as its ingress gateway sees it. Most ingress paths rewrite the address: a gateway Service with the default `externalTrafficPolicy: Cluster`, and host-level forwarding such as a host that forwards ports 80 and 443 to a `NodePort`, or a Lima VM.

If you must use literal IP addresses, see Connect by IP address (see online documentation), and preserve the caller's address:

- For a gateway Service exposed directly as a `LoadBalancer` or `NodePort`, set `externalTrafficPolicy: Local`. This does not help when host-level forwarding sits in front of the Service.
- For a load balancer in front of the gateway, enable the PROXY protocol on both the load balancer and the Istio ingress gateway, for example with the gateway annotation `proxy.istio.io/config: '{"gatewayTopology": {"proxyProtocol": {}}}'`. See Istio's PROXY protocol (https://istio.io/latest/docs/ops/configuration/traffic-management/network-topologies/#proxy-protocol) guide. Enabled on one side only, it breaks every connection through that load balancer, not only pairing.

Otherwise preflight passes but pairing fails with `source_ip_denied`. The error reports the address the receiver saw, in `observed_source`, with a remediation.

Trust the peer's certificate

Each cluster verifies the peer's edge certificate when it tests, pairs, and pings. The peer's domain must appear in the subject alternative names of the peer's edge certificate, and its issuer must be trusted:

- **Publicly trusted CA:** nothing to do.
- **Private CA:** publish every CA that signs either cluster's edge certificate, concatenated into one file, in the `kamiwaza-edge-trust` ConfigMap on **both** clusters. The simplest setup issues both edge certificates, and the shared realm's certificate, from one private CA. Create the ConfigMap before installing, as described in Edge certificate (see online documentation). If you change it on an installed cluster, restart the scheduler: `kubectl -n kamiwaza rollout restart deployment/core-scheduler`.
- **Self-signed:** federation with self-signed edge certificates is for evaluation only. Set `core.scheduler.tlsInsecure: true` in step 0 on both clusters. This turns off certificate verification on federation control calls, so never use it in production.

Pairing then exchanges and pins each cluster's platform CA for mesh traffic.

0. Add federation values to each cluster

Add these values to each cluster's values file:

Value	Set it to
`core.initialClusterName`	A name unique across the federation, for example `fed-a` and `fed-b`. Every install that leaves it unset is named `Default Cluster`.
`core.scheduler.trustedSharedIssuers`	A list holding the shared realm's issuer URL. Empty, the default, rejects every shared-realm token.
`core.delegatedJobs.enabled`, `core.delegatedJobs.admission.enabled`	`true`, on a cluster that runs federated jobs. See Job Submission (see online documentation).
`core.authz.executionGate`	An execution gate, on a cluster that runs federated jobs. Without one, the cluster refuses every job submitted from a peer. See Execution Gates (see online documentation).

On `fed-a`, for example:

```
core:
  initialClusterName: fed-a
  scheduler:
    trustedSharedIssuers:
      - https://idp.example.com/realms/federation
```

Then rerun the `helm upgrade --install` command from Install on Kubernetes with Istio (see online documentation) with the same `--version` and the updated values file. The scheduler restarts with the new values.

A later name change takes effect after the scheduler restarts, but an already-paired peer keeps its cached name until the federation is re-paired. The display name labels and selects a federation; the hostname in `remote_ips` controls the HTTP Host header and TLS SNI.

Connect by IP address

For an IP-based connection, provide both the connect address and the hostname:

```
{"ip": "10.0.0.12", "hostname": "fed-b.example.com", "primary": true}
```

Kamiwaza connects to `ip` but sends `hostname` as the HTTP Host and TLS SNI. The hostname must be the peer's domain, which its edge certificate and Gateway already cover. On the receiver, a literal IP entry is also checked against the address the pairing request arrives from; see Use hostnames for the peer address (see online documentation).

Get an admin token

The API examples use these variables. The `admin` password is the one from Verify (see online documentation):

```
A_API=https://fed-a.example.com/api
B_API=https://fed-b.example.com/api

A_ADMIN_TOKEN=$(curl --fail --silent --show-error \
  --data-urlencode username=admin --data-urlencode "password=$A_ADMIN_PASSWORD" \
  "$A_API/auth/token" | jq -er .access_token)
```

Get `B_ADMIN_TOKEN` the same way against `$B_API`. `SHARED_USER_TOKEN` is an access token for a shared-realm user, issued by the shared realm. Shared-realm tokens are often short-lived (five minutes by default in Keycloak); fetch a new one when a call returns `401`.

1. Verify both routes before pairing

In the console, select **Cluster > Federations > Add Request**, enter the peer address, and use **Test Connection**. Resolve DNS, TCP, TLS, or L7 failures before creating either record.

The API equivalent is admin-only:

```
curl --fail --silent --show-error \
  --request POST \
  --header "Authorization: Bearer $A_ADMIN_TOKEN" \
  --header 'Content-Type: application/json' \
  --data '{"remote_ips":[{"ip":"10.0.0.12","hostname":"fed-
b.example.com","primary":true}]}' \
  "$A_API/cluster/federations/preflight" | jq .
```

Run the same check in the opposite direction. Pairing needs bidirectional reachability.

2. Create the receiver record

On the receiving cluster:

1. Select **Cluster > Federations > Add Request**.
2. Choose **Wait for remote cluster**.
3. Enter the initiator's unique cluster name, and its domain as the connect address. See Use hostnames for the peer address (see online documentation).
4. Enter a strong PSK from your approved secret manager.
5. Choose **shared_idp**.
6. Enter the shared issuer URL. If the issuer uses a private CA, supply the complete CA chain. Kamiwaza derives the JWKS URL from the issuer and ignores any JWKS URL you enter.
7. Create the request.

The receiver should show a `WAITING` record.

3. Create and pair the initiator

On the initiating cluster:

1. Select **Cluster > Federations > Add Request**.
2. Choose **Initiate connection**.
3. Enter the receiver's unique name, connect address, and hostname.
4. Use **Test Connection**.
5. Enter the same PSK and shared-realm values.
6. Create the request and approve the confirmation.

Both cards should reach `PAIRED` and show **Receiver-controlled** with the same shared issuer. Use **Ping** and verify that **Last Ping** updates.

Pairing automatically exchanges the platform CA certificates. Do not update `cluster_federations.remote_ca_cert` with SQL and do not copy a raw PSK from the database.

Raw API equivalent

The console uses a two-step API:

1. `POST /api/cluster/federations` creates the receiver and initiator records.
2. `POST /api/cluster/federations/{id}/pair` on the initiator drives the handshake.

Keep the PSK in mode-`0600` request files so it does not appear in process arguments or shell history. The initiator request shape is:

```
{
  "remote_cluster_name": "fed-b",
  "remote_ips": [
    {
      "ip": "10.0.0.12",
      "hostname": "fed-b.example.com",
      "primary": true
    }
  ],
  "preshared_key": "read-from-a-private-file",
  "callback_hostname": "fed-a.example.com",
  "role": "initiator",
  "shared_issuer_url": "https://idp.example.com/realms/federation",
  "shared_ca_pem": "optional-private-ca-chain"
}
```

The receiver request uses `role: "receiver"`, the initiator's name and route, and the receiver's own domain as `callback_hostname`:

```
{
  "remote_cluster_name": "fed-a",
  "remote_ips": [
    {
      "ip": "fed-a.example.com",
      "primary": true
    }
  ],
  "preshared_key": "read-from-a-private-file",
  "callback_hostname": "fed-b.example.com",
  "role": "receiver",
  "shared_issuer_url": "https://idp.example.com/realms/federation",
  "shared_ca_pem": "optional-private-ca-chain"
}
```

Create the receiver first, then the initiator, then pair the returned initiator ID:

```
curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $B_ADMIN_TOKEN" \
  --header 'Content-Type: application/json' \
  --data-binary @receiver-create-private.json \
  "$B_API/cluster/federations" > receiver.json

curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $A_ADMIN_TOKEN" \
  --header 'Content-Type: application/json' \
  --data-binary @initiator-create-private.json \
  "$A_API/cluster/federations" > initiator.json

FEDERATION_ID="$(jq -er .id initiator.json)"
B_FEDERATION_ID="$(jq -er .id receiver.json)"
curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $A_ADMIN_TOKEN" \
  "$A_API/cluster/federations/$FEDERATION_ID/pair" | jq .
```

Delete the private request files after pairing.

4. Grant receiver-local access

Pairing establishes trust; it does not grant access to datasets, models, or jobs. On the receiver, open the federation's **Access** panel and add each shared-realm subject that may use the pair. Assign only the required receiver-local grants, for example:

- `dataset:<dataset-urn>#viewer` for discovery and retrieval;
- the exact model relation required for chat; and
- `cluster_jobs:__all__#executor` for job submission.

The API equivalent, on the receiver, uses the receiver's federation ID:

```
curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $B_ADMIN_TOKEN" \
  --header 'Content-Type: application/json' \
  --data '{
    "external_id": "<subject>@<initiator-cluster-id>",
    "initial_tuples": [
      {"subject": "user:{{user_id}}", "relation": "viewer", "object": "dataset:
<dataset-urn>"},
      {"subject": "user:{{user_id}}", "relation": "executor", "object":
"cluster_jobs:__all__"}
    ]
  }' \
  "$B_API/cluster/federations/$B_FEDERATION_ID/users" | jq .
```

For `shared_idp`, `<subject>` is the `sub` claim of the user's shared-realm token, and `<initiator-cluster-id>` is the initiating cluster's `local_cluster_id`, shown on the initiator's federation record (`GET $A_API/cluster/federations/$FEDERATION_ID`). A call from a subject that is not yet allowed fails with `403` and `unauthorized_brokered_user`; the response names the exact `external_id` to add. `{{user_id}}` is replaced with the receiver-local user ID when the receiver provisions the user on its first request. See Receiver-owned grants (see online documentation).

The receiver creates and controls the local brokered identity. Source-cluster administrator roles do not cross the federation boundary.

5. Verify the pair

Use the exact federation UUID when scripting. Name prefixes are rejected when they are ambiguous.

```
curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $A_ADMIN_TOKEN" \
  "$A_API/cluster/federations/$FEDERATION_ID/ping" | jq -e '.reachable == true'

curl --fail --silent --show-error \
  --header "Authorization: Bearer $SHARED_USER_TOKEN" \
  "$A_API/mesh/$FEDERATION_ID/api/catalog/datasets/" | jq .
```

The catalog response must contain only datasets authorized on the receiver. For up to a minute after you grant access, calls can return `503` with `authorization_unavailable` while the receiver applies the grants; retry. Continue with Federated Retrieval (see online documentation) and Job Submission (see online documentation).

Diagnose a failure

Run the stored-route diagnostic before changing records:

```
curl --fail --silent --show-error \
  --request POST --header "Authorization: Bearer $A_ADMIN_TOKEN" \
  "$A_API/cluster/federations/$FEDERATION_ID/diagnose" | jq .
```

Common stable reasons include `dns_unresolvable`, `connection_failed`, `connection_timeout`, `tls_untrusted_ca`, `tls_hostname_mismatch`, `peer_identity_aliases_local`, `psk_propagation_timeout`, `peer_ca_projection_forbidden`, and `source_ip_denied`. `source_ip_denied` appears only when pairing, after preflight has passed, and reports the address the receiver saw in `observed_source`; see Use hostnames for the peer address (see online documentation).

If CA inspection is required, support both deployed certificate layouts and never assume `root-ca` lives in the `kamiwaza` namespace:

```
if kubectl get secret core-internal-ca-tls -n kamiwaza >/dev/null 2>&1; then
  kubectl get secret core-internal-ca-tls -n kamiwaza \
    -o jsonpath='{.data.ca\.crt}' | base64 -d
else
  kubectl get secret root-ca -n kamiwaza-ca \
    -o jsonpath='{.data.ca\.crt}' | base64 -d
fi
```

This command is diagnostic only. Normal pairing exchanges and stores the CA.

Disconnect

Use **Disconnect** in the federation card or call the disconnect endpoint. Disconnect blocks new mesh admission and revokes delegated authority. Running work is reconciled through the job lifecycle; confirm terminal status before deleting evidence or receiver-local grants.

Federated Retrieval

Federated retrieval lets you discover and query data on remote clusters without the data leaving its origin. The mesh proxy routes your requests transparently — the remote cluster's catalog and retrieval services process them as if they were local.

Prerequisites

- A paired federation (see online documentation) between the clusters
- A dataset registered in the receiver's Data Catalog (see online documentation) that needs no stored credential. The S3 example on that page uses one, so federated retrieval can't read it. See Credential-backed datasets (see online documentation).
- An active receiver-side allowlist entry for the shared subject
- Explicit per-dataset ReBAC grants on the target cluster. Mesh-originated requests do not receive the native-cluster admin bypass.

Discover Remote Datasets

```
# List datasets on the remote cluster
curl -sk "https://kamiwaza.example.com/api/mesh/<remote-cluster>/api/catalog/datasets/"
\
  -H "Authorization: Bearer $TOKEN"
```

Response: array of dataset metadata (URNs, names, schemas) from the remote cluster's catalog.

Retrieve Remote Data

```
# Create a retrieval job on the remote cluster
curl -sk -X POST "https://kamiwaza.example.com/api/mesh/<remote-
cluster>/api/retrieval/jobs" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "dataset_urn": "urn:li:dataset:(urn:li:dataPlatform:file,sales_summary.csv,PROD)",
    "columns": ["region", "quarter", "revenue"],
    "limit_rows": 100,
    "transport": "inline"
  }'
```

The retrieval service on the remote cluster reads the dataset through its local data adapters and returns the results through the mesh proxy. Adapters that read a stored credential, such as S3 or Postgres with a `credential_secret`, don't work for federated callers yet; see Credential-backed datasets (see online documentation). Only the query results cross the network — raw data stays on the remote cluster.

Streaming Retrieval (SSE)

For large result sets, use the streaming transport:

```
# Submit job, then stream results
JOB_ID=$(curl -sk -X POST "https://kamiwaza.example.com/api/mesh/<remote-
cluster>/api/retrieval/jobs" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"dataset_urn": "...", "transport": "sse"}' \
  | python3 -c 'import sys,json; print(json.load(sys.stdin)["job_id"])')

# Stream results via SSE
curl -sk "https://kamiwaza.example.com/api/mesh/<remote-
cluster>/api/retrieval/jobs/$JOB_ID/stream" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Accept: text/event-stream"
```

SSE events stream through the mesh proxy transparently.

Credential-backed datasets

A federated caller can't retrieve a dataset whose connector reads a stored credential, such as an S3 bucket or a database registered with a `credential_secret`. The receiver reads the credential as the caller, and a dataset grant doesn't extend to the credential, so the retrieval fails. The same applies to a local user who can view the dataset but doesn't own the credential. Use datasets that need no stored credential for federated retrieval.

List Remote Models

```
curl -sk "https://kamiwaza.example.com/api/mesh/<remote-
cluster>/api/serving/deployments" \
  -H "Authorization: Bearer $TOKEN"
```

Returns deployment metadata: model names, engine types, status, and access paths.

Run Remote Inference

```
# Chat completion on a remote model
curl -sk -X POST \
  "https://kamiwaza.example.com/api/mesh/<remote-cluster>/runtime/models/<deployment-
id>/v1/chat/completions" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Qwen3-4B",
    "messages": [{"role": "user", "content": "Summarize the NPS themes"}]
  }'
```

The inference request routes through the mesh proxy to the remote cluster's Ray Serve deployment. The response streams back.

Access Control

Receiver-owned grants

All mesh users require explicit per-dataset ReBAC grants on the target cluster. The native-cluster admin bypass is suppressed for mesh-originated requests. The catalog listing endpoint filters results so federated users see only datasets they have receiver-local grants for.

Grant access **at federation-pairing time** through the brokered-user allowlist's `initial_tuples` field — **not** through `/api/auth/tuples`.

> ⚠ **WHY NOT `/api/auth/tuples`?**
>
> A federated caller has no local user account on the target cluster until their *first* mesh request, when the brokering service auto-provisions a local Keycloak user with a freshly-minted UUID. The per-dataset check authorizes against **that local UUID**, which is not known ahead of time — so a tuple written against the source cluster's user UUID via `/api/auth/tuples` never matches, and the call returns `204` while access stays denied (`404 Dataset not found`). The allowlist's `initial_tuples` solves this with a `{{user_id}}` placeholder that renders to the local UUID at provision time.

```
# On the TARGET cluster, add the source user to the federation allowlist
# WITH the dataset grant. `{{user_id}}` renders to the local brokered KC
# UUID when the user is auto-provisioned on first mesh request, so the
# viewer relation lands on the subject the per-dataset check looks up.
curl -sk -X POST "https://<TARGET_IP>/api/cluster/federations/<FEDERATION_ID>/users" \
  -H "Authorization: Bearer $REMOTE_ADMIN_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "external_id": "<source-user-uuid>@<source-cluster-uuid>",
    "initial_tuples": [
      {
        "subject": "user:{{user_id}}",
        "relation": "viewer",
        "object": "dataset:<dataset-urn>"
      }
    ]
  }'
```

`initial_tuples` entries use namespaced `"<namespace>:<id>"` strings; `{{user_id}}` (local brokered UUID) and `{{external_id}}` (`<source-user-uuid>@<source-cluster-uuid>`) are substituted at provision time.

`initial_tuples` are fixed when the entry is created: a second POST for the same `external_id` returns `409`, and restore cannot replace them. Include every grant the user needs when you add them. To grant more after the user has made their first request, write the tuple with `POST /api/auth/tuples` against the user's receiver-local ID, found with the `authorization_relations` query in Operations: ReBAC Denied (see online documentation).

Mesh Proxy Endpoint Reference

All federated operations use the mesh proxy at `/api/mesh/{cluster_selector}/`:

Parameter	Description
`cluster_selector`	An exact federation UUID, an exact remote cluster UUID, or the remote cluster name as an exact or prefix match. Use the federation UUID in scripts. A name or prefix that matches more than one pair returns `409 mesh_target_ambiguous`.
`{path}`	The API path on the remote cluster. Paths starting with `runtime/` are routed as-is; all others get an `/api/` prefix.

Timeouts

Path Type	Default Timeout	Env Override
API paths	60 seconds	`MESH_PROXY_READ_TIMEOUT_SECONDS`
Runtime paths (inference)	600 seconds	`MESH_PROXY_RUNTIME_READ_TIMEOUT_SECONDS`

Headers

The mesh proxy adds these headers to cross-cluster requests:

Header	Purpose
`X-KZ-Mesh-Source-Cluster-Id`	Originating cluster UUID
`X-KZ-Mesh-User-Id`	Authenticated user ID
`X-KZ-Mesh-User-Roles`	User roles (comma-separated)
`X-KZ-Mesh-Signature`	HMAC-SHA256 signature
`X-KZ-Mesh-Correlation-Id`	Request correlation ID for cross-cluster tracing
`X-KZ-Mesh-Route`	Hop trace for loop detection

IDENTITY-MODE-DEPENDENT TRUST

`X-KZ-Mesh-User-Id` / `-Roles` are **source-asserted**. In `shared_idp` (receiver-controlled) mode the receiver establishes identity from the caller's own validated shared-realm token and strips source-asserted cluster roles; the forwarded values are the identity only in source-trusted `peer_kc` mode. See Identity Trust Modes (see online documentation).

Job Submission

Submit and manage Ray jobs on local or remote clusters. Jobs run Python entrypoints on the cluster's Ray runtime and can return structured results via a log marker protocol.

Prerequisites

- For a local job: a token for a user who may run jobs on the cluster, such as `admin`.
- For a job on a peer: a shared-realm token for a subject the receiver has allowlisted with `cluster_jobs:__all__#executor`. Admin roles do not cross the federation.
- Ray cluster running on the target cluster

Federated delegated jobs additionally require receiver-side allowlisting, `cluster_jobs:__all__#executor`, exact receiver-local resource grants, an execution gate (see online documentation), and delegated jobs enabled in the receiver's values file:

```
core:
  delegatedJobs:
    enabled: true
    admission:
      enabled: true
```

Apply it by rerunning the `helm upgrade --install` command from Install on Kubernetes with Istio (see online documentation) with the updated values file.

Tenant-mode ReBAC

On an authenticated tenant-mode install, ordinary cluster jobs work with ReBAC enabled and the community fallback disabled. The request is authorized at the Kamiwaza API, then the Core API submits the job to the in-cluster Ray Dashboard Jobs API on port `8265`. The Core API is served by the Ray head workload, so the chart's Ray-head `AuthorizationPolicy` must allow the `core-ray` service account to use that port. The public ingress rule alone is not sufficient for this east-west request, and the Ray Dashboard should not be exposed as a public job-submission endpoint.

For a fresh strict-ReBAC install, verify the complete lifecycle before making the environment available to users. `ADMIN_PASSWORD` is the `admin` password from Verify (see online documentation):

```
API=https://kamiwaza.example.com/api
TOKEN=$(curl --fail --silent --show-error \
  --data-urlencode username=admin --data-urlencode "password=$ADMIN_PASSWORD" \
  "$API/auth/token" | jq -er .access_token)

BODY=$(jq -n \
  --arg ep "python -c \"import json; print('KZ_MESH_RUN_ON_JSON::' + json.dumps({'ok':
True}))\"" \
  '{entrypoint: $ep, timeout_seconds: 120}')
JOB_ID=$(curl --fail --silent --show-error -X POST "$API/cluster/jobs/submit" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d "$BODY" | jq -er .job_id)

until STATUS=$(curl --fail --silent --show-error "$API/cluster/jobs/$JOB_ID/status" \
    -H "Authorization: Bearer $TOKEN" | jq -r .status) \
  && [ "$STATUS" != PENDING ] && [ "$STATUS" != RUNNING ]; do sleep 2; done
echo "$STATUS"   # Expect SUCCEEDED

curl --fail --silent --show-error "$API/cluster/jobs/$JOB_ID/result" \
  -H "Authorization: Bearer $TOKEN" | jq .
# Expect a result of {"ok": true}
```

The check submits a small job, observes its status transition, and retrieves the structured result. A `502` containing `Ray submit failed (403)` indicates a mesh authorization policy problem on the Ray Dashboard hop; inspect the rendered `ray-head` policy and confirm the `core-ray` principal and port `8265` rule before changing ReBAC fallback settings.

Submit a Job (Async)

```
curl -sk -X POST "https://kamiwaza.example.com/api/cluster/jobs/submit" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "entrypoint": "python train.py --epochs 10",
    "runtime_env": {"env_vars": {"REPORT_FORMAT": "json"}},
    "timeout_seconds": 300
  }'
```

`runtime_env` accepts environment variables only. Kamiwaza strips Ray execution-environment keys such as `pip`, `working_dir`, `py_modules`, and `conda`. Use the approved dependency mechanism below for Python packages.

Governed delegated access and approved dependencies

A receiver-executed job must name every dataset or model operation it needs. The receiver checks the submitting identity against those exact resources before dispatch and issues renewable, job-bound authority only for that set.

Python dependencies use exact `name==version` coordinates from the receiver-owned package catalog:

```
{
  "entrypoint": "python analysis.py",
  "timeout_seconds": 900,
  "delegated_access": {
    "datasets": [
      {
        "urn": "urn:li:dataset:(urn:li:dataPlatform:postgres,nps_verbatims,PROD)",
        "operations": ["discover", "retrieve"]
      }
    ],
    "models": []
  },
  "python_packages": ["humanize==4.13.0"]
}
```

Packages are not resolved from public PyPI at request time. The operator must enable the package contract, approve each exact version and wheel SHA-256, and configure one PyPI-compatible repository. The request receives no repository URL or credential and cannot add an unapproved package.

For an air-gapped or restricted receiver, point the package installer at an operator-owned mirror by mounting a Secret containing a mode-`0600` `pip.conf`. Keep the repository URL, CA configuration, and credential out of Helm values and job payloads. Restrict the delegated driver NetworkPolicy to the repository's exact CIDRs and ports:

```
core:
  delegatedJobs:
    enabled: true
    admission:
      enabled: true
    pythonPackages:
      enabled: true
      catalog:
        - name: humanize
          version: 4.13.0
          sha256:
            - <exact-64-character-wheel-sha256>
      repository:
        existingSecret: private-pypi
      repositoryCIDRs:
        - 10.42.0.8/32
      repositoryPorts:
        - 8443
```

Kubernetes NetworkPolicy works with CIDRs, not repository DNS names. Keep the address set exact and update it when the mirror moves. A package coordinate that is absent from the catalog, has the wrong digest, or cannot be fetched fails closed before the entrypoint starts.

Response:

```
{"job_id": "uuid", "ray_job_id": "raysubmit_...", "status": "PENDING"}
```

Submit a Job to a Peer

To run a job on the receiving cluster, send the same calls through the mesh proxy on your own cluster with a shared-realm token. `$FEDERATION_ID` is the federation's ID on your cluster; see Federation setup (see online documentation). `$SHARED_USER_TOKEN` is described in Get an admin token (see online documentation).

```
MESH=https://kamiwaza.example.com/api/mesh/$FEDERATION_ID

JOB_ID=$(curl --fail --silent --show-error -X POST "$MESH/cluster/jobs/submit" \
  -H "Authorization: Bearer $SHARED_USER_TOKEN" -H "Content-Type: application/json" \
  -d "$BODY" | jq -er .job_id)

until STATUS=$(curl --fail --silent --show-error "$MESH/cluster/jobs/$JOB_ID/status" \
    -H "Authorization: Bearer $SHARED_USER_TOKEN" | jq -r .status) \
  && [ "$STATUS" != PENDING ] && [ "$STATUS" != RUNNING ]; do sleep 2; done

curl --fail --silent --show-error "$MESH/cluster/jobs/$JOB_ID/result" \
  -H "Authorization: Bearer $SHARED_USER_TOKEN" | jq .
```

`$BODY` is a job request such as the one in the lifecycle check above. The receiver applies its execution gate (see online documentation) before dispatch; a `403` with `no_execution_gate_configured_for_mesh` means the receiver has none. `/cluster/jobs/run`, `/logs`, and `/cancel` work through the mesh in the same way.

Run a Job (Synchronous)

Submits and blocks until completion, returning the extracted result:

```
curl -sk -X POST "https://kamiwaza.example.com/api/cluster/jobs/run" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "entrypoint": "python inference.py",
    "timeout_seconds": 60
  }'
```

Response:

```
{
  "job_id": "uuid",
  "ray_job_id": "raysubmit_...",
  "status": "SUCCEEDED",
  "result": {"accuracy": 0.95},
  "duration_seconds": 12.3
}
```

Check Job Status

```
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/{job_id}/status" \
  -H "Authorization: Bearer $TOKEN"
```

Returns the full job record including status, timestamps, and error details if failed.

Get Job Logs

```
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/{job_id}/logs" \
  -H "Authorization: Bearer $TOKEN"
```

Returns Ray stdout/stderr for the job.

Extract Job Result

```
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/{job_id}/result" \
  -H "Authorization: Bearer $TOKEN"
```

Extracts structured JSON from the job's stdout. The job code must print a result marker:

```
import json
result = {"accuracy": 0.95, "model": "v2"}
print(f"KZ_MESH_RUN_ON_JSON::{json.dumps(result)}")
```

Status Code	Meaning
200	Result extracted successfully
409	Job has not succeeded yet
410	Ray logs expired (result no longer available)

Cancel a Job

```
curl -sk -X POST "https://kamiwaza.example.com/api/cluster/jobs/{job_id}/cancel" \
  -H "Authorization: Bearer $TOKEN"
```

Requests cancellation of a running job. The job transitions to STOPPED status.

Job Lifecycle

Status	Description
PENDING	Submitted to Ray, not yet running
RUNNING	Executing on Ray cluster
SUCCEEDED	Completed successfully
FAILED	Exited with error
STOPPED	Canceled by user or timed out

Timeout and Auto-Cancel

Set `timeout_seconds` on submission. If the job exceeds this duration during a `/run` call, it is automatically canceled and marked with `timed_out: true`.

API Reference

Method	Path	Description
POST	`/cluster/jobs/submit`	Async submission
POST	`/cluster/jobs/run`	Sync submit + poll + result
GET	`/cluster/jobs/{id}/status`	Job status with Ray refresh
GET	`/cluster/jobs/{id}/result`	Extract structured result
GET	`/cluster/jobs/{id}/logs`	Ray stdout/stderr
POST	`/cluster/jobs/{id}/cancel`	Cancel running job

Operations and troubleshooting

Operational runbook for federated mesh operations. Covers setup verification, common failure modes, job monitoring, and diagnostic queries.

1. Pairing and first verification

Use the receiver-controlled `shared_idp` workflow in the Federation Setup (see online documentation) guide. Preflight both routes, create the receiver record first, create and pair the initiator, then add the shared subject to the receiver's allowlist. The handshake exchanges platform CA trust; do not inject certificates or PSKs with SQL.

After both cards reach `PAIRED`, verify the exact record and route. The examples use `LOCAL_API` for the local cluster's API URL, for example `https://fed-a.example.com/api`, and `LOCAL_TOKEN` for an `admin` token from Get an admin token (see online documentation):

```
# Check federation status
curl --fail --silent --show-error \
  --header "Authorization: Bearer $LOCAL_TOKEN" \
  "$LOCAL_API/cluster/federations/$FEDERATION_ID" | jq \
  '{id,status,identity_mode,trust_posture,remote_ips,shared_issuer_url}'

# Expected: "status": "PAIRED"

# Test cross-cluster connectivity by listing remote datasets
curl --fail --silent --show-error \
  --header "Authorization: Bearer $SHARED_USER_TOKEN" \
  "$LOCAL_API/mesh/$FEDERATION_ID/api/catalog/datasets/" | jq .

# Test remote model listing. Returns 403 unless the receiver granted this
# user a model relation.
curl --fail --silent --show-error \
  --header "Authorization: Bearer $SHARED_USER_TOKEN" \
  "$LOCAL_API/mesh/$FEDERATION_ID/api/serving/deployments" | jq .
```

2. Mesh Proxy Troubleshooting

Duplicate or stale cluster names

Symptoms: Federation cards show the same peer label, a renamed peer still shows its old label, a new name returns `404 mesh_target_not_found`, or a shared name or prefix returns `409 mesh_target_ambiguous`.

Diagnosis: Read the stored selector and UUID for every active federation:

```
curl --fail --silent --show-error \
  --header "Authorization: Bearer $LOCAL_TOKEN" \
  "$LOCAL_API/cluster/federations?status=PAIRED" | jq \
  '.[] | {id, remote_cluster_id, remote_cluster_name, status}'
```

`remote_cluster_name` is stored when the federation is created or paired. Renaming the peer changes its local cluster row, but ping does not refresh the cached federation row.

Resolution: Use the exact federation UUID for automation. Assign a distinct `core.initialClusterName` on every cluster, then re-pair existing federations to adopt renamed peer labels. Until re-pairing, the cached old name and federation UUID continue to select the existing pair.

HMAC Signature Failures (403)

Symptoms: 403 response on mesh proxy requests. No `rebac_denied` detail in the response body -- the request is rejected before reaching the endpoint.

Diagnosis:

1. **Run the typed route diagnostic** -- do not read PSK material from the database:

   ```
   curl --fail --silent --show-error --request POST \
     --header "Authorization: Bearer $LOCAL_TOKEN" \
     "$LOCAL_API/cluster/federations/$FEDERATION_ID/diagnose" | jq .
   ```

2. **Check clock skew** -- The HMAC signature includes a timestamp. The receiving cluster rejects signatures older than 300 seconds (configurable via `auth_forward_header_signature_ttl_seconds`):

   ```
   # Compare times on both clusters
   date -u  # Run on both clusters and compare
   ```

Resolution:

- If the diagnostic reports an HMAC or PSK failure: disconnect and re-pair both local federation records with a new out-of-band PSK. The current API does not provide an in-place PSK rotation endpoint. Never compare or copy stored secret references as if they were raw keys.
- If clock skew exceeds 300 seconds: synchronize NTP on both clusters.

ReBAC Denied (403 rebac_denied)

Symptoms: 403 with `rebac_denied` detail in the response body. The HMAC signature was valid (the request reached the endpoint), but the user lacks authorization on the target cluster.

Diagnosis:

1. Check the `authorization_relations` table on the **target** cluster. Note the relevant `subject_id` is the **local brokered KC UUID** (auto-provisioned on first mesh request), **not** the source cluster's user UUID — query by relation/object. The allowlist API does not return the local UUID; the user's

`initial_tuples` rows show it:

```
kubectl exec core-postgres-0 -n kamiwaza -- psql -U core -d kamiwaza -c \
  "SELECT subject_namespace, subject_id, relation, object_namespace, object_id
   FROM authorization_relations
   WHERE object_namespace = 'dataset' AND object_id = '<dataset-urn>';"
```

2. For catalog/retrieval operations, the remote user needs a `viewer` relation on the `dataset` namespace for each dataset they access.
3. **Admin bypass is suppressed for mesh requests.** Even if the remote user has an `admin` role on their home cluster, they still need explicit ReBAC grants on the target cluster for per-resource operations.

Resolution:

Seed the per-dataset grant through the brokered-user allowlist's `initial_tuples` field. Do **not** use `/api/auth/tuples` for a brokered user: that user has no local account on the target cluster until their first mesh request, when brokering auto-provisions a local Keycloak user with a freshly-minted UUID. The check authorizes against **that local UUID**, so a tuple written against the source UUID returns `204` but never matches (access stays `404`). `initial_tuples` resolves the `{{user_id}}` placeholder to the local UUID at provision time.

```
# Grant viewer on a dataset to a brokered user, via the allowlist. The grants
# are fixed when the entry is created; a second POST returns 409.
curl -sk -X POST "https://<TARGET_IP>/api/cluster/federations/<FEDERATION_ID>/users" \
  -H "Authorization: Bearer $REMOTE_ADMIN_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "external_id": "<source-user-uuid>@<source-cluster-uuid>",
    "initial_tuples": [
      {"subject": "user:{{user_id}}", "relation": "viewer", "object": "dataset:
<dataset-urn>"}
    ]
  }'
```

> **NO LONGER NEEDED IN 1.1**
>
> Earlier releases required a `federation:operator` relation on the source cluster to use the mesh proxy. In 1.1 the mesh egress is **authenticated-only** — that grant is no longer required, and a mesh `403` now originates from the **receiver's** per-resource ReBAC or per-record gate, not a missing source-side operator relation.

CA Trust Errors (TLS)

Symptoms: `502 mesh_proxy_bad_gateway` response. Connection refused or TLS handshake failure in the core service logs.

Diagnosis:

1. Run `POST /api/cluster/federations/{id}/diagnose` and inspect the TLS layer's stable reason. Re-pair if the handshake did not persist peer trust.

2. Verify the remote cluster's edge certificate covers the hostname in the federation's `remote_ips`:

```
# Run on the remote cluster
kubectl -n <gateway-namespace> get secret <edge-secret> -o
jsonpath='{.data.tls\.crt}' \
  | base64 -d | openssl x509 -noout -issuer -ext subjectAltName
```

3. Test TLS from the local cluster's side, with the CA you published in `kamiwaza-edge-trust`:

```
openssl s_client -connect fed-b.example.com:443 -servername fed-b.example.com \
  -CAfile ./edge-ca.crt </dev/null
```

Resolution:

- If peer trust is absent: re-pair the federation. Do not manually inject a CA.
- If the remote certificate does not cover the configured hostname, reissue it, or change the federation's hostname to one it covers.
- If the handshake reports `tls_untrusted_ca`, publish the issuing CA as described in Trust the peer's certificate (see online documentation).

Trailing Slash Redirects (307 -> auth loss)

Symptoms: 307 redirect followed by 401 or 403 on the redirected request, or a plain `403 Forbidden` on a path that works with the other slash form. The HMAC signature headers are lost during the redirect because the mesh proxy has `follow_redirects=False`.

Diagnosis:

Check the logs for 307 status codes:

```
kubectl logs deployment/core-scheduler -n kamiwaza | grep "307"
```

The HMAC signature binds to the specific request path. If the server redirects (e.g., `/api/catalog/datasets` to `/api/catalog/datasets/`), the signature becomes invalid for the new path.

Resolution:

Ensure API paths in mesh requests use the canonical form (with or without trailing slash as the endpoint expects). Most list endpoints expect a trailing slash:

```
# Correct
curl -sk "https://kamiwaza.example.com/api/mesh/studio-1/api/catalog/datasets/" ...

# May cause 307 redirect
curl -sk "https://kamiwaza.example.com/api/mesh/studio-1/api/catalog/datasets" ...
```

Clock Skew

Symptoms: Intermittent 403 errors on mesh requests. Requests succeed sometimes and fail at other times, especially near the TTL boundary.

Diagnosis:

```
# Check the time on both clusters
ssh cluster-a "date -u '+%s'"
ssh cluster-b "date -u '+%s'"
# Difference must be < 300 seconds for mesh, < 60 seconds for pairing
```

The mesh signature TTL is 300 seconds by default (`auth_forward_header_signature_ttl_seconds`). The allowed future skew is 60 seconds (`auth_forward_header_signature_allowed_future_skew_seconds`).

Resolution:

Synchronize NTP on both clusters:

```
# On each cluster node
sudo timedatectl set-ntp true
timedatectl status
```

---

3. Job Monitoring

Job States

Status	Description
`PENDING`	Submitted to Ray, queued for execution
`RUNNING`	Actively executing on the Ray cluster
`SUCCEEDED`	Completed successfully
`FAILED`	Exited with an error
`STOPPED`	Canceled by user or timed out

Checking Job Status

Local job:

```
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/<job-id>/status" \
  -H "Authorization: Bearer $TOKEN" | python3 -m json.tool
```

Federated job (submitted via mesh):

```
curl -sk "https://kamiwaza.example.com/api/mesh/studio-1/cluster/jobs/<job-id>/status"
\
  -H "Authorization: Bearer $TOKEN" | python3 -m json.tool
```

Checking Job Logs

```
# Local job logs
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/<job-id>/logs" \
  -H "Authorization: Bearer $TOKEN" | python3 -m json.tool

# Federated job logs
curl -sk "https://kamiwaza.example.com/api/mesh/studio-1/cluster/jobs/<job-id>/logs" \
  -H "Authorization: Bearer $TOKEN" | python3 -m json.tool
```

Result Extraction

Jobs return structured results via a log marker protocol. The job code prints a marker to stdout:

```
import json
result = {"accuracy": 0.95, "model": "v2"}
print(f"KZ_MESH_RUN_ON_JSON::{json.dumps(result)}")
```

The platform scans the job's Ray logs (reverse, last match wins) for `KZ_MESH_RUN_ON_JSON::` and parses the JSON payload that follows.

Extract the result:

```
curl -sk "https://kamiwaza.example.com/api/cluster/jobs/<job-id>/result" \
  -H "Authorization: Bearer $TOKEN"
```

Error responses:

Status Code	Meaning
200	Result extracted successfully
409 Conflict	Job has not reached `SUCCEEDED` status yet
410 Gone	Ray logs have expired or the `KZ_MESH_RUN_ON_JSON::` marker was not found

Timeout and Auto-Cancel

When `timeout_seconds` is set on job submission and the job is executed via the synchronous `/run` endpoint, the platform polls Ray until the timeout elapses. If the job is still running at that point:

1. The job is canceled via the Ray Dashboard API
2. The `timed_out` flag is set to `true` in the `federated_jobs` table
3. The job status transitions to `STOPPED`

The timeout applies only during synchronous `/run` calls. Async `/submit` does not enforce timeouts automatically.

---

4. Database Queries

Useful `psql` queries for monitoring federation state. Connect via:

```
kubectl exec core-postgres-0 -n kamiwaza -- psql -U core -d kamiwaza
```

List all federations and their status

```
SELECT id, remote_cluster_name, status, last_ping,
       CASE WHEN remote_ca_cert IS NOT NULL THEN 'yes' ELSE 'NO' END AS has_ca_cert,
       created_at
FROM cluster_federations
ORDER BY created_at DESC;
```

List all federated jobs

```
SELECT id, status, source, source_cluster_id, source_cluster_name,
       user_id, timed_out, submitted_at, ended_at
FROM federated_jobs
ORDER BY submitted_at DESC
LIMIT 20;
```

List mesh-originated jobs only

```
SELECT id, status, source_cluster_name, user_id,
       entrypoint, timed_out, error_type
FROM federated_jobs
WHERE source = 'mesh'
ORDER BY submitted_at DESC;
```

Check ReBAC relations for federation

```
SELECT subject_namespace, subject_id, relation, object_namespace, object_id
FROM authorization_relations
WHERE object_namespace = 'federation'
ORDER BY created_at DESC;
```

Find failed jobs with error details

```
SELECT id, source_cluster_name, entrypoint,
       error_type, error_message, ended_at
FROM federated_jobs
WHERE status = 'FAILED'
ORDER BY ended_at DESC
LIMIT 10;
```

Check for timed-out jobs

```
SELECT id, source, source_cluster_name, entrypoint,
       timeout_seconds, submitted_at, ended_at
FROM federated_jobs
WHERE timed_out = true
ORDER BY ended_at DESC;
```

5. Useful Headers for Debugging

All cross-cluster mesh requests include these headers. They are set by the mesh proxy on the source cluster and verified by ext-authz on the target cluster.

Header	Purpose
`X-KZ-Mesh-Source-Cluster-Id`	UUID of the originating cluster. Used to look up the federation record and PSK on the receiving side.
`X-KZ-Mesh-User-Id`	Source-asserted user id. Whether the target adopts it as the identity is identity-mode-dependent (see note).
`X-KZ-Mesh-User-Roles`	Source-asserted roles (e.g., `admin,editor`). Source-asserted cluster roles are stripped for receiver-controlled modes.
`X-KZ-Mesh-Signature`	HMAC-SHA256 signature over the canonical payload (cluster ID, user ID, roles, route, method, URI).
`X-KZ-Mesh-Signature-Ts`	Unix timestamp (seconds) when the signature was issued. Must be within TTL window (default 300s).
`X-KZ-Mesh-Route`	Comma-separated list of cluster IDs this request has traversed. Used for loop detection (max 8 hops).
`X-KZ-Mesh-Correlation-Id`	UUID for cross-cluster request tracing. Preserved across hops; generated on the first hop if absent.

IDENTITY-MODE-DEPENDENT TRUST

In `shared_idp` (receiver-controlled) mode the target establishes identity from the caller's **own validated shared-realm token**, not the source-asserted `X-KZ-Mesh-User-Id` / `-Roles`. The source-asserted values are the identity only in source-trusted `peer_kc` /grandfathered mode. See Identity Trust Modes (see online documentation).

Signature Payload

The HMAC signature binds to these fields (all lowercased in the canonical form):

Field	Source
`x-kz-mesh-source-cluster-id`	Source cluster UUID
`x-kz-mesh-user-id`	Authenticated user ID
`x-kz-mesh-user-roles`	Comma-separated roles
`x-kz-mesh-route`	Hop trace
`x-forwarded-method`	HTTP method (uppercased)
`x-forwarded-uri`	Request path (query string stripped)

Changing any of these fields after signing invalidates the signature. This prevents replay attacks across different endpoints or methods.

Execution Gates

The federated job service on the receiving cluster consults a pluggable **execution gate** before dispatching any job to Ray. The gate decides whether this user is allowed to run jobs on this cluster, independent of what data the job will read. Record-level filtering — "what columns and rows can this user see once the job runs?" — is handled separately by per-dataset attribute gates (see the retrieval docs).

This page covers the operator-facing surface: what the default behavior is, how to configure a policy, and how to diagnose 403 responses on federated job submissions.

Default behavior

An Execution Gate is a cluster-wide resource: one gate is in force at a time. When no gate is configured, the platform applies **source-scoped default policy**:

Submission source	Default when unconfigured	Reason code
`local` (same-cluster authenticated user)	**ALLOW**	`no_execution_gate_configured_local_allowed`
`mesh` (arrived via federation peer)	**DENY (403)**	`no_execution_gate_configured_for_mesh`

Local submissions preserve the pre-feature behavior of single-cluster clusters. Mesh submissions — the new cross-cluster surface — are denied until an operator explicitly declares the cluster's policy. This prevents a forgotten configuration from silently accepting federated work.

A gate value that is set but does not parse is not treated as unconfigured. Every submission, local or mesh, is refused with `gate_config_invalid` until the value is fixed; see Troubleshooting (see online documentation).

Configuring a gate

A gate can be set in three places. The cluster enforces the first one that holds a value:

1. **The API.** `PUT /api/cluster/execution-gate` binds a gate at runtime. It applies to the next submission, with no restart.
2. **The `AUTHZ_EXECUTION_GATE` environment variable.** Use it only for ad hoc testing.
3. **The Helm value** `core.authz.executionGate.{type, config}`, which the chart renders into the `core-runtime-config` ConfigMap.

Use one mechanism per cluster, so the active gate is not in doubt. `GET /api/cluster/execution-gate` reports the active gate and, in `source`, which of the three it came from: `api`, `env`, or `helm`. See Verifying the active gate (see online documentation).

Quick start: permit all mesh submissions

> **⚠ WARNING**
>
> `AllowAllExecutionGate` admits every authenticated federated submission. A mesh job submitted without `delegated_access` runs on the receiving cluster's shared Ray head, next to the platform's own services. Use it only for evaluation clusters and peers you trust. For any other peer, keep the default mesh-deny, or write a site-specific gate that admits only the peers and users you trust. A gate sees the job's `source`, `source_cluster_id`, `user_id`, `entrypoint` and `target_datasets`, but not whether it carries `delegated_access`. See Federated Job Submission (see online documentation) for delegated jobs.

If your cluster's security posture is "accept all authenticated federated submissions" (for example, an evaluation cluster or a single-tenant deployment), set the classpath to the shipped allow-all gate. Add this to the cluster's values file:

```
core:
  authz:
    executionGate:
      type: kamiwaza.services.authz.gates.default_gates.AllowAllExecutionGate
      config: {}
```

Then rerun the `helm upgrade --install` command from Install on Kubernetes with Istio (see online documentation) with the same `--version` and the updated values file. The chart rolls the pods when the rendered configuration changes, and the explicit allow-all decision replaces the default mesh-deny policy.

An administrator can also bind a gate at runtime through the API, without a Helm upgrade. A gate bound through the API takes precedence over the Helm value until it is cleared; see Clearing an API binding (see online documentation). `ADMIN_TOKEN` is an `admin` token for the cluster, obtained as in Get an admin token (see online documentation):

```
curl --fail --silent --show-error \
  --request PUT --header "Authorization: Bearer $ADMIN_TOKEN" \
  --header 'Content-Type: application/json' \
  --data
'{"type":"kamiwaza.services.authz.gates.default_gates.AllowAllExecutionGate","config":
{}}' \
  "https://kamiwaza.example.com/api/cluster/execution-gate" | jq .
```

Installing a site-specific policy

For clusters with a real policy — clearance gating, tenant isolation, program compartments, etc. — point at your own gate class. The platform does **not** ship site-specific gates. Ship your class in one of two ways:

- **As a gate package (see online documentation).** An administrator installs it through the API, and Kamiwaza loads its gate classes with no further configuration. Gate-package storage is off by default; that page says how to turn it on.
- **In your own build of the Core image.** Also list the class's package in `AUTH_GATE_CLASSPATH_PREFIXES`, as described in Allow your gate's classpath (see online documentation). From 1.3.4, Kamiwaza refuses to load a class that is not listed.

Then set the classpath, for example in your values file:

```
core:
  authz:
    executionGate:
      type: acme_sec.policy.ClearanceExecutionGate
      config:
        min_clearance: C
        require_authenticated: true
```

Requirements for the class:

- Its module must be importable on the Ray head, `core-raycluster-head`. The Kamiwaza API runs there: it evaluates the gate on every job submission and serves the binding API. A gate package is mounted there.
- Its classpath must be allowed; see Allow your gate's classpath (see online documentation).
- It must subclass `kamiwaza.services.authz.gates.ExecutionGate` and implement `name` (property), `required_attributes()`, and `authorize()`.
- The gate is created with the `config` dict. A changed `config` takes effect like any other gate change; see When a gate change takes effect (see online documentation).

Gate authors' guide: see the Kamiwaza Agent SDK docs for the `AttributeGate` / `ExecutionGate` plugin authoring interface.

Allow your gate's classpath

Kamiwaza imports a gate class only when its classpath:

- starts with `kamiwaza.services.authz.gates.` or `kamiwaza_extensions.`,
- belongs to an installed gate package (see online documentation), or
- starts with a prefix listed in `AUTH_GATE_CLASSPATH_PREFIXES`.

This applies to the execution gate and to dataset gates, whether the gate is set through the API, the `AUTHZ_EXECUTION_GATE` environment variable, or the Helm value. Kamiwaza checks the classpath before it imports anything. Before 1.3.4, only the API checked it, so a class set through the environment variable or the Helm value loaded without the check. From 1.3.4, a class that fails it is refused:

- Every job submission, local and mesh, returns `403` with reason `gate_load_error`.
- `GET /api/cluster/execution-gate` returns `403` with reason `classpath_not_allowed`. Earlier 1.3 releases return the same, so you can check before you upgrade.
- A dataset gated by such a class is not served: retrieval returns `503`, and a job result that reads the dataset returns `403`.

Classes from an installed gate package need nothing more. For a class in your own build of the Core image:

1. Choose the prefix: the name of the package that holds your gates, ending with a dot. For `acme_sec.policy.ClearanceExecutionGate`, use `acme_sec.`. A prefix allows every module whose name starts with it, so keep it as narrow as your package. Separate several prefixes with commas.

2. Add it to your values file:

```
core:
  scheduler:
    extraEnv:
      - name: AUTH_GATE_CLASSPATH_PREFIXES
        value: "acme_sec."
```

`core.scheduler.extraEnv` sets the variable on `core-scheduler`, on the Ray head and workers, and on `core-api` if you enabled it. It is a list, and Helm replaces a list as a whole: if any of your values files already sets `core.scheduler.extraEnv`, add this entry to that list and keep the entries already in it. If you enabled the retrieval streamer (`core.retrievalStreamer.enabled: true`), add the same entry to `core.retrievalStreamer.extraEnv`.

3. Rerun the `helm upgrade --install` command from your environment's install page, with the same `--version` and your updated values files. The Core pods restart, which interrupts the API and any job running on the Ray head. Before an upgrade to 1.3.4 or later, you can instead add the entry first and run the upgrade: it applies the entry.

4. Verify. As an administrator, `GET /api/cluster/execution-gate` returns `200`, with your class in `type`:

   ```
   curl --silent --show-error --header "Authorization: Bearer $ADMIN_TOKEN" \
     "https://kamiwaza.example.com/api/cluster/execution-gate" | jq .
   ```

   For each gated dataset, `GET /api/catalog/datasets/<URL-encoded URN>/gate` returns `200`.

Env var override (ad-hoc testing)

For one-off testing or debugging without a Helm rollout, you can set the env var on the Ray head, `core-raycluster-head`. The Kamiwaza API runs there: it enforces the gate on job submissions and reports it through `GET`. Setting it on `core-scheduler` has no effect. Setting or removing it restarts the Ray head, which interrupts the API and any job running there, so don't do it on a cluster that is serving work:

```
kubectl set env deployment/core-raycluster-head -n kamiwaza \

AUTHZ_EXECUTION_GATE='{"type":"kamiwaza.services.authz.gates.default_gates.AllowAllExecutio
{}}'
```

The value is a JSON string. It takes precedence over the Helm value, but not over a gate bound through the API, and `GET /api/cluster/execution-gate` reports it with `source: env`.

The variable is stored in the Deployment's pod template, so restarts and rollouts keep it, and it keeps overriding the Helm value. Do not rely on a `helm upgrade` to remove it. Remove it explicitly when you are done:

```
kubectl set env deployment/core-raycluster-head -n kamiwaza \
  AUTHZ_EXECUTION_GATE-
```

Then check that `GET /api/cluster/execution-gate` no longer reports `source: env`; see Verifying the active gate (see online documentation).

Verifying the active gate

Ask the cluster which gate is active:

```
curl --silent --show-error --header "Authorization: Bearer $ADMIN_TOKEN" \
  "https://kamiwaza.example.com/api/cluster/execution-gate" | jq .
```

Expected when a gate is configured:

```
{
  "type": "kamiwaza.services.authz.gates.default_gates.AllowAllExecutionGate",
  "config": {},
  "gate_name": "allow_all_execution_gate",
  "kind": "execution",
  "source": "helm"
}
```

`source` names where the gate came from: `api`, `env`, or `helm`. When no gate is configured anywhere, the call returns `404` with reason `not_configured`. That is the correct state for the "default mesh-deny" posture.

When the highest layer that holds a value holds one that does not parse, the call returns `409` with reason `execution_gate_invalid`, the `source` that holds it, and a remediation. A lower layer's gate does not apply in its place.

Clearing an API binding

`DELETE /api/cluster/execution-gate` removes a gate bound through the API. A gate from the env var or the Helm value, if there is one, then applies again, and the response reports it in `active_after`:

```
{
  "deleted": true,
  "previous_type": "kamiwaza.services.authz.gates.default_gates.AllowAllExecutionGate",
  "active_after": {
    "type": "acme_sec.policy.ClearanceExecutionGate",
    "source": "helm"
  }
}
```

`active_after` is `null` when no gate remains. The API cannot remove a gate that comes only from the env var or the Helm value: `DELETE` then returns `409` with reason `execution_gate_not_api_managed`. Change the Helm value or remove the env var instead.

The Kamiwaza API logs the active gate's decision on every job submit. It writes to Ray's session logs on the Ray head, which `kubectl logs` does not show. To read them:

```
kubectl -n kamiwaza exec deploy/core-raycluster-head -- \
  grep -rh "Execution gate" /app/tmp/ray/session_latest/logs
```

Each decision line contains. for example, `Execution gate allow_all_execution_gate decision: allow (reason=default_allow_all)`.

Troubleshooting

403 on a federated job submit

The API returns a structured body with the gate's reason code:

```
{
  "detail": {
    "reason": "no_execution_gate_configured_for_mesh",
    "gate": "(unconfigured)"
  }
}
```

Common reason codes:

`reason`	Meaning	Fix
`no_execution_gate_configured_for_mesh`	Receiving cluster has no execution gate set and the default policy denies mesh submissions.	Configure a gate on the rece cluster; see Configuring a ga online documentation). If you s value, `GET /api/cluster/ gate` should report it with `s helm`.
`gate_config_invalid`	A gate value is set but does not parse, so every submission is refused, local ones included.	`GET /api/cluster/execu` returns `409` with reason `execution_gate_invalid` `source` that holds the valu value, or clear an API bindin `DELETE`.
`gate_load_error`	The configured class cannot be loaded: its classpath is not allowed, or its module cannot be imported.	If `GET /api/cluster/exec gate` returns `403` with rea `classpath_not_allowed`, your gate's classpath (see on documentation). Otherwise fin error in the Kamiwaza API's Ray head, which `kubectl` not show: `kubectl -n kam deploy/core-raycluster-h grep -rh "Failed to loa execution gate" /app/tmp/ray/session_la` Make sure the module is imp the Ray head.
`gate_exception`	The gate's `authorize()` method raised.	Find the traceback in the Ka API's log on the Ray head: `kamiwaza exec deploy/co raycluster-head -- grep "raised during authoriz /app/tmp/ray/session_la` Fix the gate, or unset the co to defaults.
(site-specific)	Reason strings beyond the ones above come from the configured gate itself (e.g., `clearance_insufficient`, `tenant_mismatch`).	Look at the gate's own docs reason vocabulary.

When a gate change takes effect

The gate is resolved on every job submission, so a gate bound or cleared through the API applies to the next one. A Helm value change takes effect when `helm upgrade` rolls the pods, which the chart does whenever the rendered configuration changes. An env var change takes effect when the pods restart.

Ray worker pods do not evaluate the gate — it runs once per submit at the control plane, before Ray sees the job.

Gate configured but local submissions still bypass

That is correct behavior. Local submissions (from same-cluster authenticated users) also run through the configured gate — if your gate `authorize()` method returns ALLOW for them, they pass. If you want the gate to be the sole gatekeeper for both local and mesh, your gate's `authorize()` should inspect `job_spec["source"]` and apply identical rules to both. The platform's *default* policy is source-scoped only when no gate is configured.

See also

- Operations & Troubleshooting (see online documentation) — general federation ops runbook, including federation-pair setup and diagnostic queries.
- Federated Job Submission (see online documentation) — the client side of the `/cluster/jobs/` API.
- Federation API Reference (see online documentation) — full endpoint reference.

Gate Packages

Gate packages let you install custom **ExecutionGate** and **AttributeGate** classes onto a paired cluster from a Python package index — without rebuilding the platform image. This capability is available in Kamiwaza 1.2.0.

A gate package is an ordinary Python package whose modules contribute one or more gate classes (subclasses of `ExecutionGate` or `AttributeGate`). The cluster's `kz.gates.packages` API (Python module: `kamiwaza_sdk.gates.packages`) installs the package into an isolated per-package directory, makes its classes reachable from `kz.gates.discover`, and persists the install record. The classes are then bindable via the existing `cluster.set_execution_gate(...)` and `datasets.set_gate(...)` APIs covered in Execution Gates (see online documentation).

When to use this

Use gate packages when:

- You need to apply policy logic (clearance gating, attribute mapping, tier-based access) that lives in code your team owns
- You want to ship that policy without forking or rebuilding the Kamiwaza platform image
- Your gate class implements either the `ExecutionGate` protocol (cluster-scoped job authorization) or `AttributeGate` protocol (dataset-scoped attribute resolution)

Do not use gate packages for:

- General-purpose application code — gate packages are isolated to the gate runtime and do not have access to the broader application context
- Policy that is already expressible via standard ReBAC relations — use those directly via the `subjects` and `datasets` APIs instead

Lifecycle overview

```
install ─▶ discover ─▶ bind ─▶ (job submit / dataset access) ─▶ replace ─▶ uninstall
                                                                  │
                                                                  └─ atomic in-place
                                                                     (no unbound-gate
window)
```

Each lifecycle action is an SDK call:

Action	SDK call	Required permission
Install	`kz.gates.packages.install(spec, hash_digest=..., index_url=...)`	admin
List installed	`kz.gates.packages.list()`	admin
Get one	`kz.gates.packages.get(name)`	admin
Atomic replace	`kz.gates.packages.replace(name, new_spec, hash_digest=..., index_url=...)`	admin
Uninstall	`kz.gates.packages.uninstall(name)`	admin (refused if any active binding references the package's classpaths)
Discover a class	`kz.gates.discover(classpath)`	viewer or above
Bind as ExecutionGate	`kz.cluster.set_execution_gate(type=..., config=...)`	admin
Bind as AttributeGate	`kz.datasets.set_gate(urn=..., type=..., config=...)`	admin

Installing a gate package

```python
from kamiwaza_sdk import KamiwazaClient

kz = KamiwazaClient(base_url="https://kamiwaza.example.com/api")

# Hash-pin to the exact wheel you've verified out-of-band.
# pip refuses installs whose downloaded artifact doesn't match.
result = kz.gates.packages.install(
    "acme-gates==1.0.0",
    hash_digest="sha256:<sha256-of-the-wheel>",
    index_url="https://pypi.example.com/simple",
)
print(result.package.name, result.package.version)
print(result.package.classpaths)  # ['acme_gates.gate.AcmeAttributeGate', ...]
```

The install path:

1. Resolves the wheel from `index_url` (any PEP 503 simple index is acceptable, including PyPI, a private mirror, or a local file index served over HTTP)
2. Validates the SHA-256 digest against the supplied `hash_digest` via `pip install --require-hashes` — if pip cannot verify, install fails before any files are written
3. Installs into a per-package directory under the gate-packages PVC (mounted on the Ray head, the scheduler, and the API server pods)

4. Scans the installed package for `ExecutionGate` / `AttributeGate` subclasses and records the resulting classpaths in the `cluster_gate_packages` table
5. Extends the API server's classpath allowlist and `sys.path` so `kz.gates.discover` sees the new gate immediately

Discovering and binding

Once installed, the gate classpath is reachable via discover:

```
gate = kz.gates.discover("acme_gates.gate.AcmeAttributeGate")
print(gate.name, gate.kind, gate.config_schema)
```

Bind as a cluster `ExecutionGate` (if the class implements that protocol) or as a per-dataset `AttributeGate` (if the class implements that protocol). The same APIs work whether the class came from a built-in module or an installed gate package — there is no separate binding path for installed gates.

Atomic replace

Replace updates the package in place without leaving a window where the bound gate class is unimportable. The platform installs the new version into a staging sibling directory, validates that the new version's classpath set is a *superset* of the old version's bound classpaths, then performs a two-rename POSIX swap.

```
result = kz.gates.packages.replace(
    "acme-gates",
    "acme-gates==1.0.1",
    hash_digest="sha256:<sha256-of-the-new-wheel>",
    index_url="https://pypi.example.com/simple",
)
```

If the new version drops a classpath that is currently bound, the replace is refused with `GatePackageClasspathDropError` and the live tree is unchanged. This is the **classpath-superset guarantee**: an actively-bound gate can never be silently removed by an upgrade.

Hash-mismatch refusal

The platform refuses an install or replace whose downloaded artifact does not match the supplied hash:

```
from kamiwaza_sdk.exceptions import GatePackageHashMismatchError

try:
    kz.gates.packages.replace(
        "acme-gates",
        "acme-gates==1.0.1",
        hash_digest="sha256:" + "0" * 64,  # wrong on purpose
        index_url="https://pypi.example.com/simple",
    )
except GatePackageHashMismatchError as exc:
    print(exc.body["detail"])
```

The refusal is enforced by pip's `--require-hashes` mode; the platform does not compute or trust a separately-derived hash. The live tree is untouched.

Uninstall

```
kz.gates.packages.uninstall("acme-gates")
```

The uninstall path checks for active bindings: any cluster `ExecutionGate` or per-dataset `AttributeGate` whose classpath matches one contributed by the package will cause the uninstall to refuse with `GatePackageUninstallBlockedError`, naming the bound classpath(s). Unbind the gate first, then retry.

Security model

The gate-packages feature ships with several boundaries:

- **Hash pinning** is required on every install and replace. The platform does not allow unhashed installs.
- **Admin-only API**: the install/replace/uninstall endpoints require the `admin` role on the local cluster. Federation peers cannot install gate packages remotely.
- **NetworkPolicy isolation**: when the optional top-level chart value `workerNetworkPolicy.enabled` is set to `true`, the Ray worker pods running gate code are blocked from arbitrary internet egress. The Ray head pod has a similar policy under the top-level `rayHeadNetworkPolicy.enabled`. Both are opt-in; default off. (Note the chart keys are top-level, not nested under `worker.networkPolicy` or `rayHead.networkPolicy` — a common operator override mistake that silently no-ops.)
- **Audit trail**: every install/replace/uninstall emits a `gate_package_lifecycle` audit event with `actor`, `action`, `name`, `version`, `hash_digest`, and (for replace) `prior_hash`.

Limits and trade-offs

- A gate-package install affects the local cluster only. Federated peers must install the same package on their side if they want to bind it locally.

- The classpath-superset guarantee means rollback to an older package version is an *install* (of the older version) plus an *unbind/rebind*, not an automatic operation. Plan upgrades to be additive when possible.
- The PVC backing gate-packages-venv ships **disabled by default** (`authz.gatePackages.pvc.enabled: false` in the platform chart) — operators must explicitly opt in before any gate-package install will succeed. Once enabled, the PVC falls through to `ReadWriteOnce`, which is sufficient on single-node clusters. Multi-node clusters must override **both** `authz.gatePackages.pvc.accessModes` to `[ReadWriteMany]` AND `authz.gatePackages.pvc.storageClassName` to a RWX-Filesystem class (CephFS / NFS / similar) — overriding only one silently no-ops. See your install team's helm overrides documentation.

Reference

- **APIs:** see API Reference (see online documentation) for the exact request/response shapes.
- **Gate classes:** see Execution Gates (see online documentation) for the `ExecutionGate` and `AttributeGate` protocols a package contributes.

Federation API reference

Reference for the federation-management and mesh-proxy endpoints. User-facing routes accept a Keycloak JWT or personal access token (PAT). Federation management writes require a native-realm administrator; a mesh-forwarded credential cannot manage a receiver's federation records.

Federation management

Manage cluster pairing, trust metadata, and brokered-user admission.

List federations

```
GET /api/cluster/federations
```

Returns all federation records to an authenticated native-realm caller. This route does not apply a per-user federation ReBAC filter. The optional `status` query parameter accepts `PAIRING`, `PAIRED`, `DISCONNECTED`, or `DELETED`; `skip` and `limit` provide pagination.

Response:

```
[
  {
    "id": "uuid",
    "remote_cluster_name": "string",
    "remote_ips": [{"ip": "string", "hostname": "string", "primary": true}],
    "callback_hostname": "string",
    "local_cluster_id": "uuid",
    "remote_cluster_id": "uuid",
    "status": "WAITING | PAIRING | PAIRED | DISCONNECTED | DELETED",
    "last_ping": "timestamp",
    "created_at": "timestamp",
    "identity_mode": "shared_idp | peer_kc",
    "trust_posture": "receiver_controlled_shared | source_trusted",
    "grandfathered": false,
    "brokering_enabled": true
  }
]
```

Get a federation

```
GET /api/cluster/federations/{federation_id}
```

Returns one federation record by UUID. This route requires a native-realm administrator and returns `404` when the record does not exist.

Create and pair a federation

```
POST /api/cluster/federations
POST /api/cluster/federations/{federation_id}/pair
```

Create the receiver's `WAITING` record first, then the initiator record. Call `/{federation_id}/pair` on the initiator to drive the signed handshake and CA exchange. Both endpoints require a native-realm administrator.

Request:

```
{
  "remote_cluster_name": "fed-b",
  "remote_ips": [
    {"ip": "10.0.0.12", "hostname": "fed-b.example.internal", "primary": true}
  ],
  "preshared_key": "read-from-private-input",
  "role": "initiator",
  "shared_issuer_url": "https://idp.example.internal/realms/federation"
}
```

Update a federation

```
PUT /api/cluster/federations/{federation_id}
```

Updates only the fields that are safe to change in place:

- `brokering_enabled`
- `shared_ca_pem`
- `require_peer_jwt`

`shared_jwks_url` is also accepted but has no effect: the receiver derives the JWKS URL from the shared issuer.

The endpoint rejects PSK, peer-binding, issuer-binding, identity-mode, remote address, lifecycle, and audit-field changes. Delete and re-pair the federation to change its PSK, peer binding, shared issuer, or identity mode.

Request:

```
{
  "brokering_enabled": true,
  "require_peer_jwt": true
}
```

Disconnect a federation

```
POST /api/cluster/federations/{federation_id}/disconnect
POST /api/cluster/federations/{federation_id}/disconnect?force=true
```

The default form asks the peer to disconnect before the local record enters `DISCONNECTED`. The `force=true` form performs the local transition when the peer cannot acknowledge the request. Both forms require a native-realm administrator.

Delete a federation

```
DELETE /api/cluster/federations/{federation_id}
```

Soft-deletes the local federation record and performs best-effort peer and identity-provider teardown. Use the pairing flow to create a new record after deletion.

Manage allowlisted users

```
POST /api/cluster/federations/{federation_id}/users
GET /api/cluster/federations/{federation_id}/users
POST /api/cluster/federations/{federation_id}/users/{external_id}/revoke
POST /api/cluster/federations/{federation_id}/users/{external_id}/restore
```

These native-administrator routes manage the receiver's brokered-user allowlist. `external_id` identifies the source subject. Optional `initial_tuples` are rendered when the receiver provisions the local brokered user; `{{user_id}}` expands to that receiver-local UUID. Revoke disables new mesh admission. The optional `cancel_in_flight_jobs=true` query parameter also marks the user's active jobs stopped on a best-effort basis. Restore re-enables the allowlist entry and can replace its tuple recipe.

Add or update request:

```
{
  "external_id": "<source-user-uuid>@<source-cluster-uuid>",
  "initial_tuples": [
    {
      "subject": "user:{{user_id}}",
      "relation": "viewer",
      "object": "dataset:<dataset-urn>"
    }
  ]
}
```

Ping a federation

```
POST /api/cluster/federations/{federation_id}/ping
```

Tests authenticated end-to-end connectivity through the signed peer path and returns `{ "federation_id": "...", "reachable": true }` on success.

Preflight and diagnose

```
POST /api/cluster/federations/preflight
POST /api/cluster/federations/{federation_id}/diagnose
POST /api/cluster/federations/resolve-address
```

Preflight probes a proposed route before persistence. Diagnose probes the stored route. Resolve-address derives the IP/FQDN complement used by the console pairing wizard. These endpoints require a native-realm administrator.

Mesh Proxy

The mesh proxy forwards requests to remote federated clusters. Every request is HMAC-signed and ReBAC-gated.

Proxy Path Pattern

```
{METHOD} /api/mesh/{federation_selector}/{remote_path}
```

`{federation_selector}` accepts an exact federation UUID, an exact remote cluster UUID, or the federation record's `remote_cluster_name` as an exact or prefix match. Use the federation UUID for automation. A name or prefix that matches multiple paired rows returns `409 mesh_target_ambiguous` instead of choosing a peer.

`remote_cluster_name` is the peer label stored when the federation is created or paired. Ping does not refresh it after a peer rename. The cached old name and federation UUID continue to work; re-pairing adopts the new display name.

`{remote_path}` is the path on the remote cluster. The proxy adds `/api` when the path does not already begin with `/api` or `/runtime`.

Authorization: Mesh egress is **authenticated-only** — any authenticated local user may call `/api/mesh/{fed}/*`. There is **no** `federation:operator` gate on the egress path (it was removed in 1.1). Cross-cluster authorization is decided entirely by the **receiving** cluster (its allowlist, per-resource ReBAC, and per-record gates).

Request headers forwarded upstream:

Header	Source	Purpose
`X-KZ-Mesh-Source-Cluster-Id`	Local cluster ID	Identifies the originating cluster
`X-KZ-Mesh-User-Id`	Local user's `sub` claim	Source-asserted user id (see note)
`X-KZ-Mesh-User-Roles`	Local user's roles (CSV)	Source-asserted roles (see note)
`X-KZ-Mesh-Route`	`{method} {path}`	Bound into the HMAC signature
`X-KZ-Mesh-Signature`	HMAC-SHA256	Verified on the remote cluster
`X-KZ-Mesh-Signature-Ts`	Unix timestamp	Replay protection (5-minute window)
`X-KZ-Mesh-Correlation-Id`	Per-request UUID	Observability tracing
`X-KZ-Mesh-Peer-Token`	Caller's bearer token	Validated by the receiver in receiver-controlled identity modes and hashed into the signed envelope
`X-KZ-Mesh-User-Attributes`	Source's `X-User-Attributes`	Source-asserted attributes (ignored in receiver-controlled modes)

> **⚠ IDENTITY-MODE-DEPENDENT TRUST**
>
> Whether the receiver **trusts** the source-asserted identity headers (`X-KZ-Mesh-User-Id` / `-Roles` / `-Attributes`) depends on the federation's identity mode (see online documentation). In `shared_idp` (receiver-controlled) the receiver establishes identity and attributes from the caller's **own validated shared-realm token**, strips source-asserted cluster roles, and **ignores** the source attribute header (F10 — "shared identity ≠ shared authority"). The source-asserted headers are trusted only in source-trusted `peer_kc` / grandfathered mode.

Stripped before forwarding:

- `Authorization` (the value is carried in the dedicated peer-token field, not as the receiver's normal bearer header)
- `Cookie`
- `Proxy-*` headers

Common Proxied Endpoints

Path	Purpose
`GET /api/mesh/{fed}/catalog/datasets/`	List remote datasets
`GET /api/mesh/{fed}/serving/deployments`	List remote models
`POST /api/mesh/{fed}/retrieval/jobs`	Run a retrieval job on the remote cluster
`GET /api/mesh/{fed}/retrieval/jobs/{id}/stream`	SSE stream of a remote retrieval
`POST /api/mesh/{fed}/cluster/jobs/run`	Submit and run a Ray job on the remote cluster

Error Responses

Status	Condition
`401`	Local auth failed (invalid JWT / PAT)
`403` `rebac_denied`	The **receiver's** per-resource ReBAC blocks the operation (there is no source-side `federation:operator` egress gate in 1.1)
`400` / `404`	Invalid route, local target, or no exact paired federation
`502` `mesh_proxy_bad_gateway`	Receiver connection, TLS, or upstream gateway failure
`504` `mesh_proxy_timeout`	Remote request exceeded the proxy timeout
`508`	Mesh hop/loop limit exceeded

Job Submission

Submit and manage Ray jobs on local or remote clusters. For the full lifecycle narrative with examples, see Job Submission (see online documentation).

Submit Asynchronously

```
POST /api/cluster/jobs/submit
```

Submits a job to Ray and returns immediately. Poll `/status` or `/result` to track it.

Request:

```
{
  "cluster_selector": "local" | "federation_name",
  "entrypoint": "python analysis.py",
  "runtime_env": {
    "env_vars": {"KEY": "value"}
  },
  "metadata": {"label": "value"},
  "timeout_seconds": 300,
  "delegated_access": {
    "datasets": [
      {
        "urn": "urn:li:dataset:(urn:li:dataPlatform:postgres,orders,PROD)",
        "operations": ["discover", "retrieve"]
      }
    ],
    "models": []
  },
  "python_packages": ["humanize==4.13.0"]
}
```

`runtime_env` accepts environment variables only. Delegated jobs may request exact package versions from the receiver's approved package catalog; arbitrary Ray `pip`, `working_dir`, `py_modules`, and `conda` settings are stripped.

Response:

```
{
  "id": "uuid",
  "ray_job_id": "string",
  "status": "PENDING",
  "cluster_selector": "string",
  "submitted_at": "timestamp"
}
```

Run Synchronously

```
POST /api/cluster/jobs/run
```

Submits the job, polls until completion (or timeout), extracts the result marker, and returns everything in one response.

Response:

```
{
  "id": "uuid",
  "status": "SUCCEEDED" | "FAILED" | "STOPPED",
  "result": { ... },
  "duration_seconds": 3.1,
  "timed_out": false,
  "error_message": "string | null"
}
```

Get Status

```
GET /api/cluster/jobs/{job_id}/status
```

Get Result

```
GET /api/cluster/jobs/{job_id}/result
```

Returns the structured result extracted from the job's log marker (`KZ_MESH_RUN_ON_JSON::{...}`).

Get Logs

```
GET /api/cluster/jobs/{job_id}/logs
```

Returns Ray stdout/stderr for the job.

Cancel

```
POST /api/cluster/jobs/{job_id}/cancel
```

Signals Ray to cancel a running job.

ReBAC Relations

Fine-grained authorization. All mesh and federation operations go through ReBAC checks.

Grant a Relation

```
POST /api/auth/tuples
```

Request:

```
{
  "subject": {"namespace": "user", "id": "<uuid>"},
  "relation": "operator" | "viewer" | "owner" | "executor",
  "object": {
    "namespace": "federation" | "dataset" | "model" | "cluster_jobs",
    "id": "string"
  }
}
```

This grants a relation to a **local** user — one with an account on the cluster you call. It does **not** work for brokered mesh users (see the note below).

Common Grant Patterns

Scenario	Namespace	Relation	Notes
Manage a federation (admin)	`federation`	`operator`	Federation **management** endpoints. **Not** required to call `/api/mesh/{fed}/*` — mesh egress is authenticated-only in 1.1
Local user can query a dataset	`dataset`	`viewer`	For native (non-mesh) retrieval on this cluster
User can submit jobs	`cluster_jobs`	`executor`	Object id is the constant `"__all__"`
User can own a dataset	`dataset`	`owner`	Can write and manage

> **BROKERED MESH USERS**
>
> A federated caller has **no local account** on the target cluster until their first mesh request, when brokering auto-provisions a local Keycloak user with a freshly-minted UUID. The per-dataset check authorizes against that local UUID, so granting `dataset:viewer` via `/api/auth/tuples` with the source UUID returns `204` but never matches (retrieval stays `404`). Grant cross-mesh dataset access through the federation allowlist's `initial_tuples` instead — see Retrieval → Access Control (see online documentation).

Attribute Headers

User attributes flow through ext-authz from JWT claims to domain gates on the retrieval service.

> **CROSS-CLUSTER ATTRIBUTES**
>
> On a **cross-cluster (mesh)** call, the receiver does not blindly forward the source's attributes. In `shared_idp` mode it derives `X-User-Attributes` from the caller's **own validated shared-realm token** and ignores the source's forwarded `X-KZ-Mesh-User-Attributes` (F10); the source-forwarded header is trusted only in source-trusted `peer_kc` mode. See Identity Trust Modes (see online documentation).

X-User-Attributes

Generic JSON header carrying all custom Keycloak user attributes. Set by ext-authz on the gateway when the JWT has custom claims (e.g. `clearance`, `country`, `department`).

Example:

```
X-User-Attributes: {"clearance":"S","country":"GBR"}
```

Forwarded across the mesh as `X-KZ-Mesh-User-Attributes`. Attribute gates on the retrieval service read this header to filter records per-user.

See the Classification Gate design (see online documentation) for the full attribute gate architecture.

Federation 1.3.0 release notes

Since Kamiwaza 1.2.0, execution and attribute gates return reduced authorization metadata. The change establishes a privacy boundary: a denied caller can learn that its view was filtered, but cannot learn the size of the withheld set or the policy that made the decision. The sections before execution gate configuration (see online documentation) describe that change for clients that still expect the earlier metadata. The execution gate configuration change is new in 1.3.0.

`gate_audit` footer compatibility

Before 1.2.0, a gated result could expose counts and the gate name:

```
{
  "gate_audit": {
    "gate": "acme-clearance",
    "included": 4,
    "redacted": 1,
    "total": 5
  }
}
```

Since 1.2.0 the four existing keys remain present for structural compatibility, but their values are always `null`:

```
{
  "gate_audit": {
    "filtered": true,
    "included": null,
    "redacted": null,
    "total": null,
    "gate": null
  }
}
```

`filtered` is the only disclosure signal. It is `true` when rows were withheld or when released rows were narrowed to fewer columns. Column narrowing is not row redaction, so it contributes zero to `redacted_count` in the counting contract below.

The old keys are intentionally not numeric zeroes. Existing consumers that index a kev continue to receive `None`: consumers that perform arithmetic or ordering (for example, `metadata["redacted"] > 0`) now fail with a `TypeError` and must migrate to the released rows plus `filtered`.

Envelope shape is seam-specific

Do not infer the shape from the runtime type. On the job-result seam, `metadata["gate_audit"]` is a **list** with one entry per gated target. On inline and SSE retrieval seams it is a **dict** for the current gate. This asymmetry is retained for compatibility; branch on the API seam documented for your client.

For chained gates, each list entry describes that gate's own input. A later entry can therefore be `filtered: false` even when an earlier gate already removed rows. A caller asking whether the overall view changed should read the aggregate `metadata["filtered"]` flag, not infer it from one entry.

GateResult counting contract

Gate authors return a `GateResult`, but the runner owns the release boundary and validates the result before any records leave the cluster. For an input batch of `N` records:

- `included_count == len(records)` — the records returned by the gate are the records released;
- `redacted_count` counts records withheld **entirely**; column narrowing is not counted as row redaction; and
- `included_count + redacted_count == total_count`.

Counts are converted with Python's `operator.index`, so integer-like values such as NumPy integer scalars are accepted while floats, strings, and other non-integral values are rejected. A gate result that violates any invariant is replaced with a whole-batch fail-closed result and the audit reason is `gate_postcondition_violation`. No partially trusted records from that batch are released.

Gate authors should test both the ordinary pass/filter path and malformed postconditions. The runner replaces a malformed result with a whole-batch fail-closed result before any records leave the cluster.

SDK and application migration

The SDK's MiniClearance acceptance helper now preserves every parsed footer, including explicit `null` values, and its assertions count the rows actually released. It must not sum or compare the four deprecated fields. The offline consumer contract is covered by `tests/integration/test_mini_clearance_gate_audit_contract.py` and the parser is documented in `tests/integration/_mini_clearance.py`.

Applications should:

1. treat `filtered` as the disclosure-safe indicator;
2. count the records/columns in the response they actually received;
3. tolerate the retained null keys during the compatibility window; and
4. avoid using a gate name or a withheld-count estimate for authorization or UI decisions.

Deprecation window

The four keys (`included`, `redacted`, `total`, and `gate`) are retained as nullable fields for the 1.3.x compatibility window. They are candidates for removal in the next breaking API/schema revision after downstream SDK and UI consumers have removed numeric and policy-name dependencies. Until that window is announced, clients must accept the keys when present and must not require them to be numeric. `filtered` and the released payload are the stable contract.

Execution gate configuration

1.3.0 also changes how the cluster finds its execution gate. Check these before upgrading:

- **The Helm value takes effect.** Earlier releases ignored `core.authz.executionGate`, so a cluster that set it ran with no gate until an administrator bound one through the API. After the upgrade, a gate set there applies.
- **Precedence.** A gate bound through the API wins, then the `AUTHZ_EXECUTION_GATE` environment variable, then the Helm value. `GET /api/cluster/execution-gate` reports the gate in force and, in `source`, where it came from.
- `DELETE` **clears only an API binding.** It returns `409` with reason `execution_gate_not_api_managed` for a gate from the Helm value or the env var.
- **A value that does not parse refuses every job.** It no longer falls back to the unconfigured default, which allows local jobs. Submissions fail with `gate_config_invalid`, and `GET` returns `409` with reason `execution_gate_invalid`.

See Execution Gates (see online documentation).

See also Execution Gates (see online documentation) for gate configuration and Federation API Reference (see online documentation) for the transport envelopes.

Models Overview

Kamiwaza provides a comprehensive system for managing the entire lifecycle of your AI models, from discovery and download to deployment and serving. This guide walks you through the key concepts and processes for working with models on the Kamiwaza platform.

Key Concepts

Kamiwaza is integrated directly with the Hugging Face Hub, allowing you to access a vast collection of open-source models. Models are identified by their Hugging Face repository ID, such as `meta-llama/Llama-3.3-70B-Instruct`.

Choosing the Right Model

Selecting the right model and configuration is crucial for achieving optimal performance and efficiency. The Kamiwaza platform automatically selects the best serving engine for your hardware and model type, but understanding the options will help you make informed decisions.

Model Formats and Engine Compatibility

Kamiwaza supports several model formats, each best suited for different serving engines and hardware configurations:

- **GGUF**: These models are highly optimized for CPU inference and are the standard for the `llama.cpp` engine. They are ideal for running on consumer hardware, including laptops with Apple Silicon, and support various quantization levels to reduce memory requirements.
- **Safetensors**: This is a safe and fast format for storing and loading tensors. On macOS with Apple Silicon, `.safetensors` models are best served by the `MLX` engine to take full advantage of the GPU. On Linux with NVIDIA, AMD and other supported GPUs or accelerators (for example, Intel Gaudi 3), they are typically served with `vLLM`.
- **Other formats (PyTorch, etc.)**: General-purpose models are typically served using `vLLM` on servers equipped with NVIDIA or AMD GPUs.

Model Serving Engines

Kamiwaza intelligently routes model deployment requests to the most appropriate serving engine. Here are the primary engines available in the platform:

vLLM Engine

- **Purpose**: Designed for high-throughput, low-latency LLM serving on powerful GPUs.
- **Best For**: Production environments with dedicated accelerators, such as NVIDIA, Intel Gaudi HPUs, or AMD GPUs.
- **Key Features**:
  - **PagedAttention**: An advanced attention algorithm that dramatically reduces memory waste.

    - **Continuous Batching**: Batches incoming requests on the fly for better GPU utilization.
    - **Tensor Parallelism**: Distributes large models across multiple GPUs.

llama.cpp Engine

- **Purpose**: Optimized for efficient CPU-based inference and a popular choice for running models on consumer hardware.
- **Best For**:
    - Running models on machines without a dedicated high-end GPU.
    - Local development on both Intel-based and Apple Silicon Macs.
- **Key Features**:
    - **GGUF Format**: Uses GGUF format which supports various levels of quantization for memory efficiency.
    - **Cross-Platform**: Runs on Linux, macOS, and Windows.
    - **Metal Acceleration**: On macOS, uses the Apple Silicon GPU for acceleration.

MLX Engine

- **Purpose**: Specifically built to take full advantage of Apple Silicon (M series) chips.
- **Best For**: High-performance inference on modern Mac computers.
- **Key Features**:
    - **Unified Memory**: Leverages the unified memory architecture of Apple Silicon for efficient data handling.
    - **Native Process**: Runs as a native macOS process, not in a container, for direct hardware access.
    - **Vision-Language Models**: Supports multi-modal models.

Ampere llama.cpp Engine

- **Purpose**: A specialized variant of `llama.cpp` optimized for Ampere arm-based CPU architectures.
- **Best For**: Running GGUF models on Ampere CPUs, such as the AmpereOne M servers.

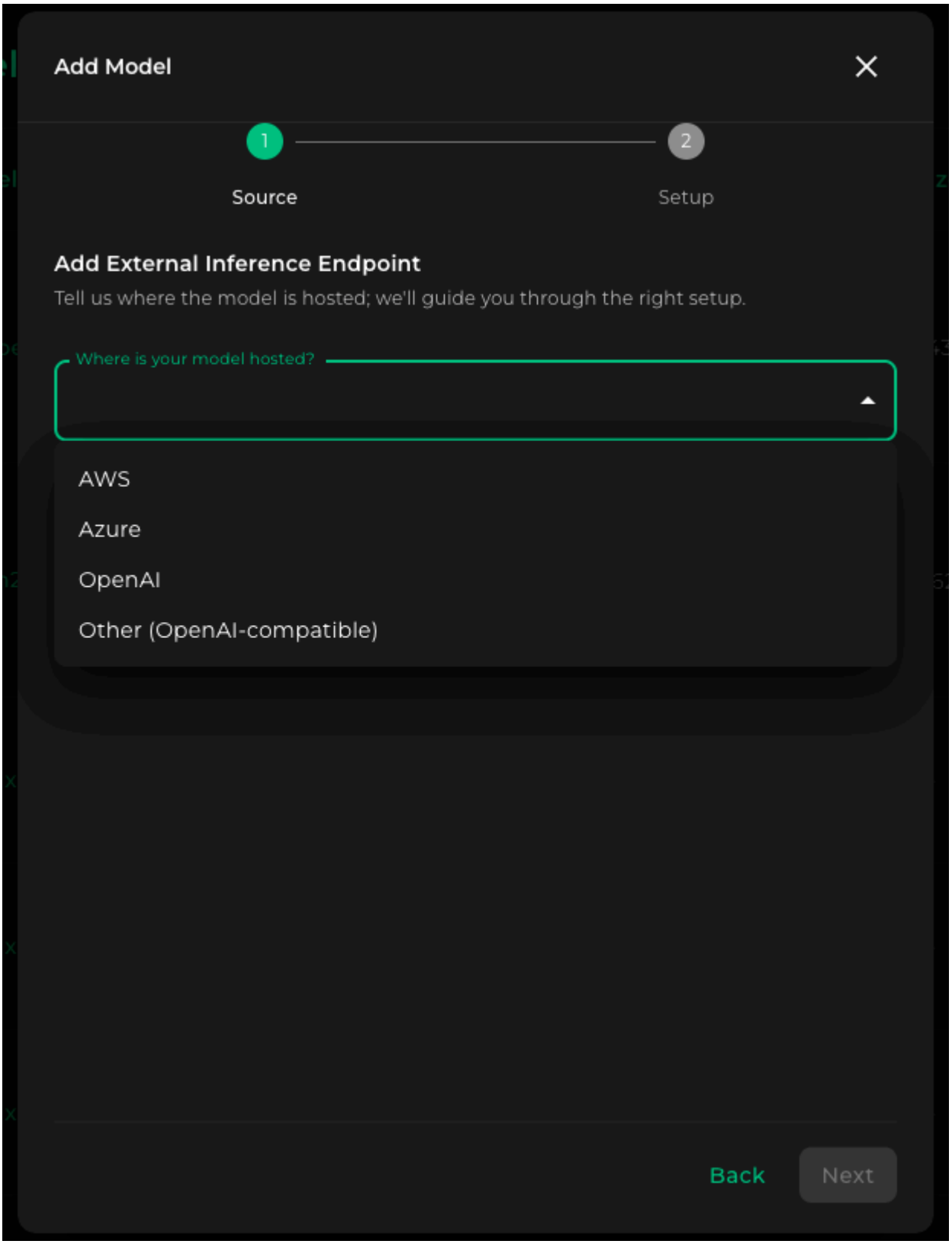
External Endpoints

In addition to running models locally, Kamiwaza can register external inference endpoints — either cloud-hosted services or customer-operated proxies — and expose them through the same deployment, audit, and access-control surfaces as local models.

Supported integrations:

- **AWS Bedrock** (see online documentation) — Anthropic Claude, Meta Llama, Amazon Nova, and other Bedrock-hosted families.
- **AWS Transcribe** (see online documentation) — managed batch and streaming speech-to-text.
- **OpenAI-compatible chat** (see online documentation) — OpenAI directly, Azure OpenAI Service, Azure AI Foundry, customer-hosted LiteLLM proxies, and any other provider whose chat API matches the OpenAI shape.
- **OpenAI-compatible transcription** (see online documentation) — OpenAI Whisper, Azure Whisper / gpt-4o-transcribe, and any other provider whose transcription API matches the OpenAI shape.

Registration is an admin action: an operator pastes provider credentials into the registration form, and once deployed, the endpoint is invoked the same way as any local deployment. The Source step of the wizard offers four hosting options — AWS, Azure, OpenAI, and Other (OpenAI-compatible) — and routes to a provider-specific Setup form from there. See each provider page for the exact wizard fields.



Credentials catalog

Credentials registered through any of the four flows above are encrypted on save and stored in the Kamiwaza secret catalog under a deterministic key derived from the provider and target — AWS region for Bedrock and Transcribe, base URL for OpenAI-compatible chat and transcription.

When you register a second endpoint that targets the same key — a second Bedrock model in the same region, or a second OpenAI-compatible model on the same base URL — Kamiwaza recognizes the existing credential and offers:

- **Use existing credential** — keep the stored value. No new entry is created.
- **Use a different credential** — overwrite. The replacement propagates to every endpoint sharing that credential within the engine credential cache window (roughly five minutes), without restarting any deployment.

You can rotate the credential later from any model's **Edit** form; rotation propagates the same way. When a model is deleted, Kamiwaza cleans up secrets that no other model still references.

> **Use long-lived credentials.** Temporary or session credentials expire while a deployment is running and lead to silent authentication failures partway through use.

Deploying Models in Novice Mode

Novice mode is Kamiwaza's default deployment mode designed to simplify the model deployment experience for new users and common use cases.

What is Novice Mode?

Novice Mode is the default, streamlined experience for selecting and deploying models. It uses a curated Model Guide to suggest strong defaults for common tasks and automatically picks a platform-appropriate variant (GPU, Mac, or CPU) with sensible settings so you can deploy quickly without deep configuration.

Key Features

- **Curated recommendations**: Short, high-quality list with clear descriptions, use cases, and scores.
- **Platform-aware variants**: Automatically selects the right build for your hardware with VRAM guidance.
- **One-click deploy**: Starts a model server with good defaults for context length and KV cache.
- **Safe defaults**: Minimal choices up front; advanced options are hidden by default.
- **Easy exit ramp**: Switch to Advanced Mode any time for full control.

Getting Started

1. Open the Models page in the Kamiwaza UI (Novice Mode is on by default for new users).
2. Browse the recommended list and pick a model that matches your task (chat, coding, reasoning, etc.).
3. Review the suggested variant for your hardware and click Deploy.
4. Test the endpoint or open the built-in chat to verify it is running.
5. If the catalog looks empty or outdated, refresh the page or restart the server to reload the guide.

When to Use Novice Mode

- **Fast start** without learning every configuration knob
- **Standard workflows**: general chat, coding help, reasoning, data analysis
- **Limited hardware**: laptops, single-GPU boxes, or CPU-only environments
- **Demos/classrooms** where reliability and simplicity matter

Advanced Configuration

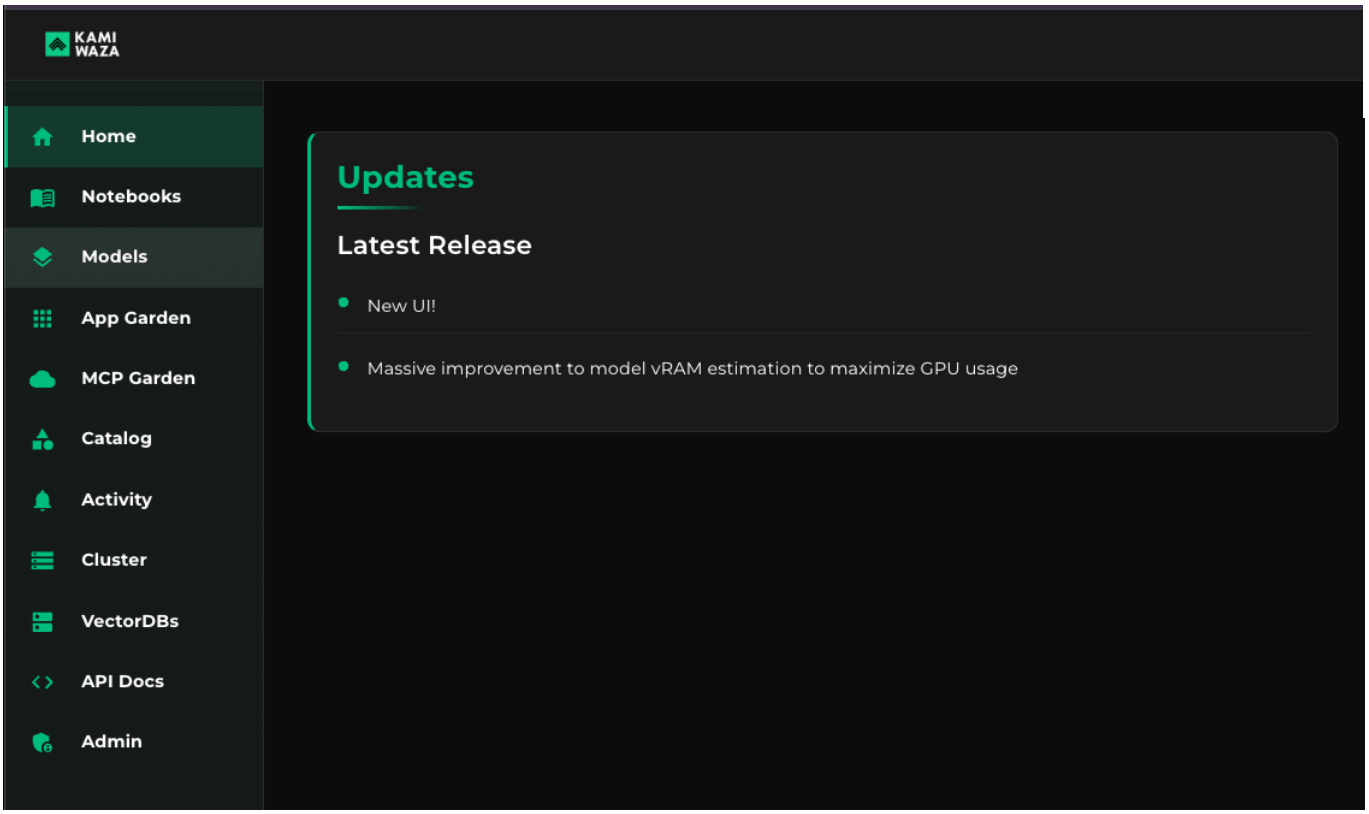
Need more control? Switch to Advanced Mode from settings to customize deployment parameters (quantization, memory, batching, prompt formatting, and more). Helpful guides:

- Deploying Models (see online documentation)
- Downloading Models (see online documentation)
- GUI Walkthrough (see online documentation)

GUI Walkthrough

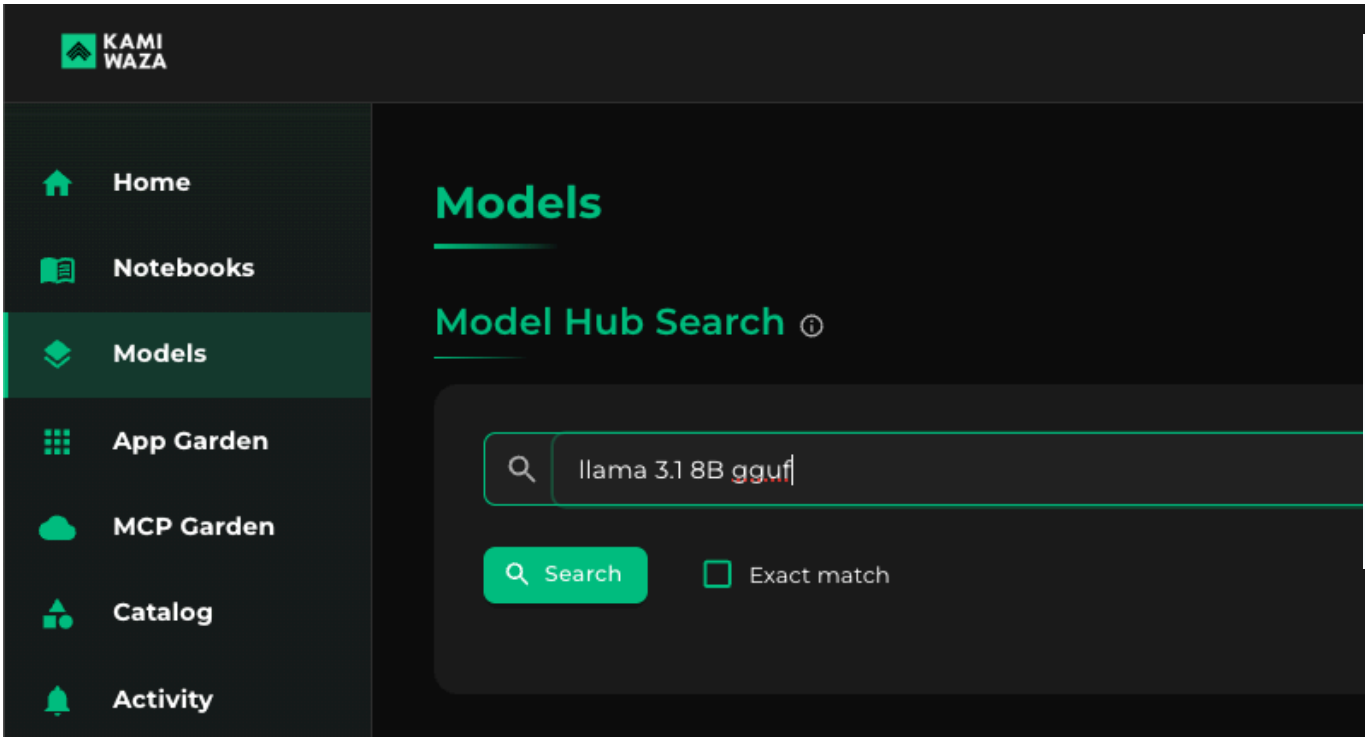
The following walkthrough is based on the Kamiwaza 1.0.0 user interface. It covers navigating the Kamiwaza admin console to **search** for a model, **download** model files, and then **deploy and use** an inference endpoint.

Step 1: Find and click the Models menu in the sidebar

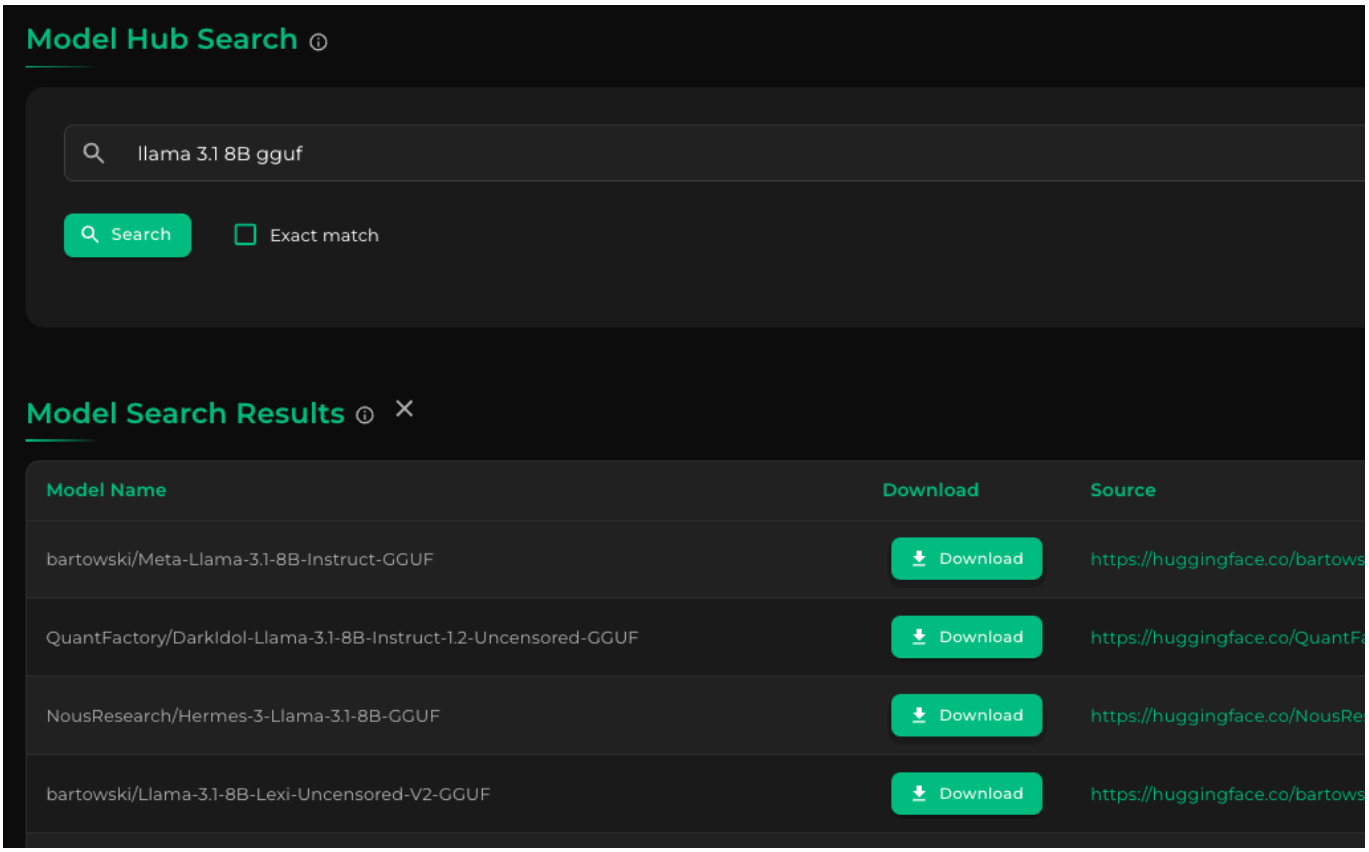


Step 2: Under Model Hub Search, type keywords for your desired model

This example looks for a GGUF of Llama3.1 8B Instruct, so typing a few of its keywords narrows the results.

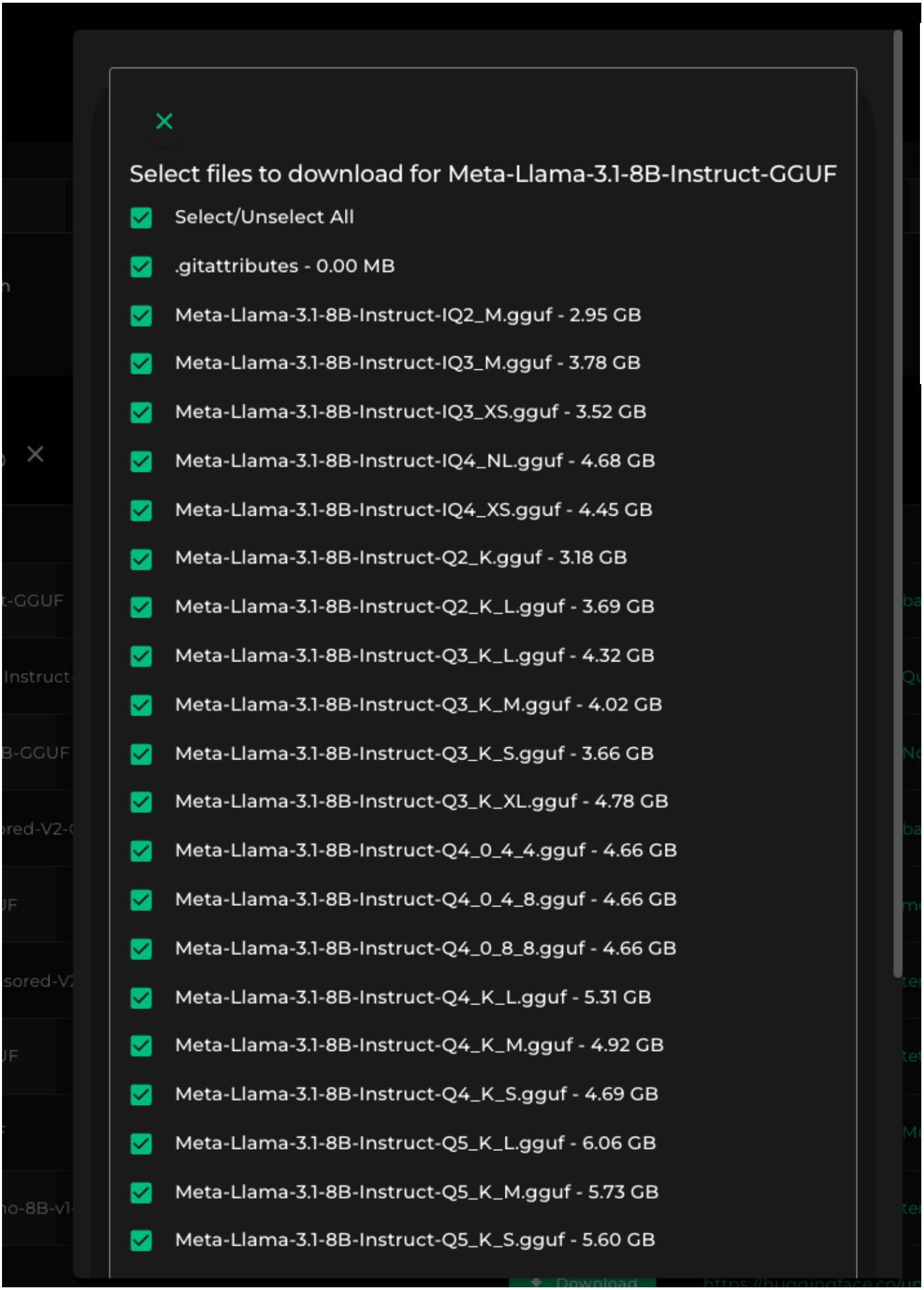


Clicking the **Search** button will show the results, like this:



Step 3: Download models files from chosen model

From the results, choose the Bartowski GGUF. Clicking the **Download** button for *bartowski/Meta-Llama-3.1-8B-Instruct-GGUF* results in the following:


Select files to download for Meta-Llama-3.1-8B-Instruct-GGUF
Select/Unselect All
.gitattributes - 0.00 MB
Meta-Llama-3.1-8B-Instruct-IQ2_M.gguf - 2.95 GB
Meta-Llama-3.1-8B-Instruct-IQ3_M.gguf - 3.78 GB
Meta-Llama-3.1-8B-Instruct-IQ3_XS.gguf - 3.52 GB
Meta-Llama-3.1-8B-Instruct-IQ4_NL.gguf - 4.68 GB
Meta-Llama-3.1-8B-Instruct-IQ4_XS.gguf - 4.45 GB
Meta-Llama-3.1-8B-Instruct-Q2_K.gguf - 3.18 GB
Meta-Llama-3.1-8B-Instruct-Q2_K_L.gguf - 3.69 GB
Meta-Llama-3.1-8B-Instruct-Q3_K_L.gguf - 4.32 GB
Meta-Llama-3.1-8B-Instruct-Q3_K_M.gguf - 4.02 GB
Meta-Llama-3.1-8B-Instruct-Q3_K_S.gguf - 3.66 GB
Meta-Llama-3.1-8B-Instruct-Q3_K_XL.gguf - 4.78 GB
Meta-Llama-3.1-8B-Instruct-Q4_0_4_4.gguf - 4.66 GB
Meta-Llama-3.1-8B-Instruct-Q4_0_4_8.gguf - 4.66 GB
Meta-Llama-3.1-8B-Instruct-Q4_0_8_8.gguf - 4.66 GB
Meta-Llama-3.1-8B-Instruct-Q4_K_L.gguf - 5.31 GB
Meta-Llama-3.1-8B-Instruct-Q4_K_M.gguf - 4.92 GB
Meta-Llama-3.1-8B-Instruct-Q4_K_S.gguf - 4.69 GB
Meta-Llama-3.1-8B-Instruct-Q5_K_L.gguf - 6.06 GB
Meta-Llama-3.1-8B-Instruct-Q5_K_M.gguf - 5.73 GB
Meta-Llama-3.1-8B-Instruct-Q5_K_S.gguf - 5.60 GB

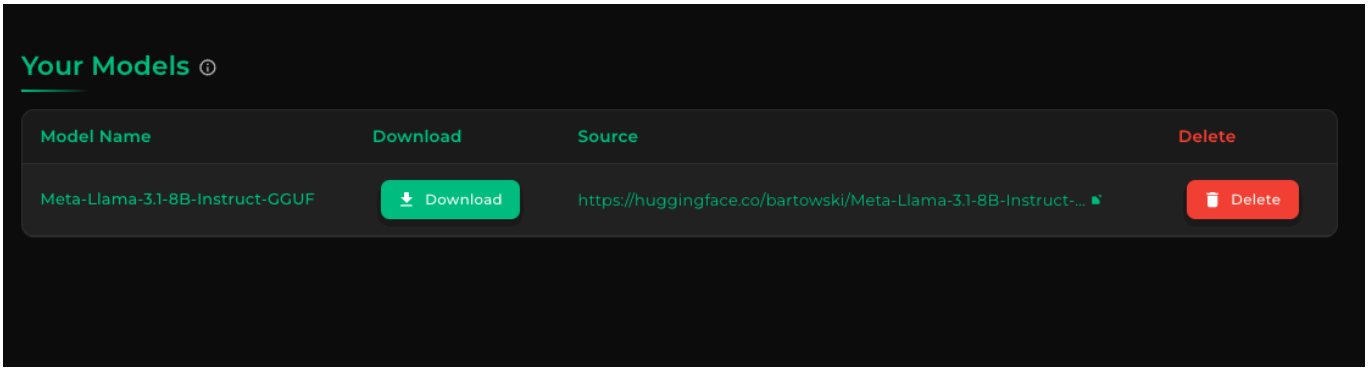
This example downloads GGUF models for a llamacpp inference engine deployment. Unlike normal Hugging Face models (safetensors), only one or two files are required — just the specific quantized model.

For this example, uncheck everything and choose only the **Q8_0** variant. Click **Select/Unselect"** all to uncheck all files, then scroll down and click the file marked **Q8_0** (near the very bottom), and then click **Download Selected Files** to download the 8-bit quantized version of Llama 3.1 8B Instruct in GGUF format.

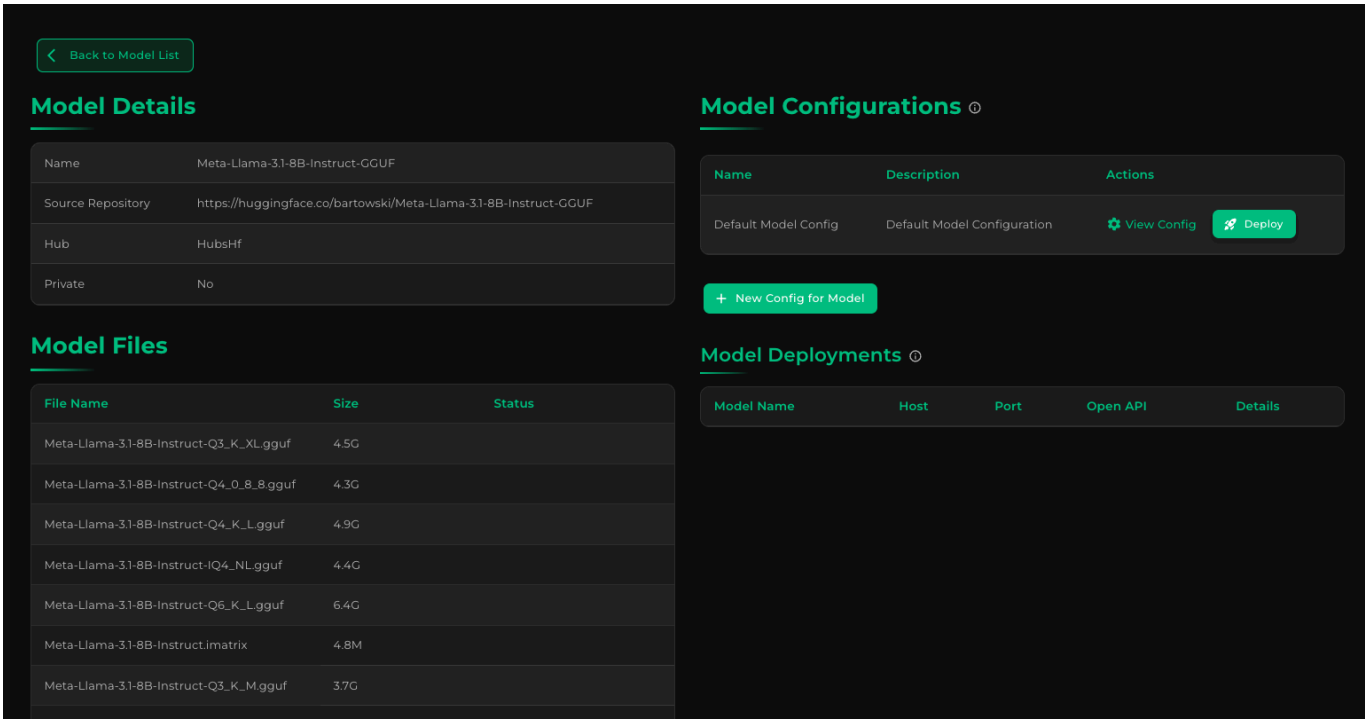
> NOTE: Normal Hugging Face models (non-GGUF) usually require all files in order to serve a model (safetensors, config, tokenizer, and so on), which is why everything is pre-checked. In that case, scroll down and click the **Download Selected Files** button.

Step 4: Check the model files after the download

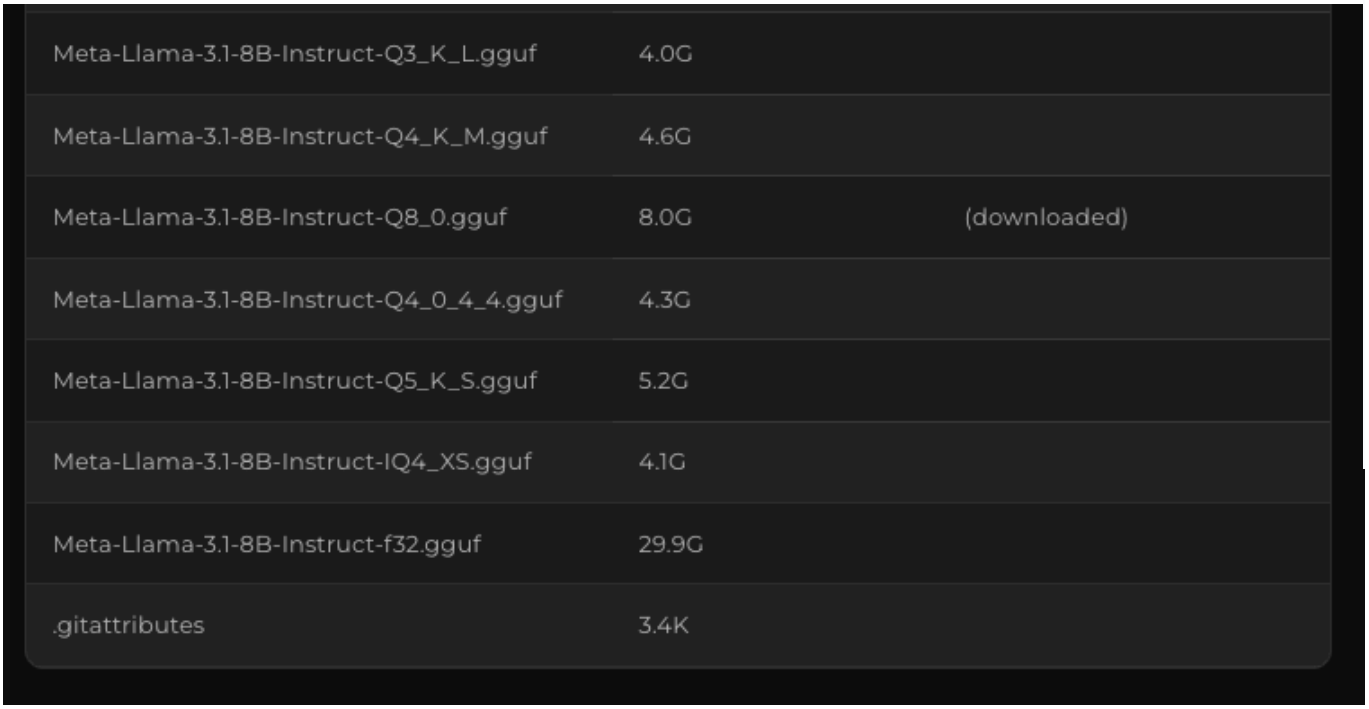
After the download, your Kamiwaza models console will look something like this:



Your downloaded model is displayed. To view the specific files downloaded for this model, click the name of the model in the list. You will see a screen that looks like this:



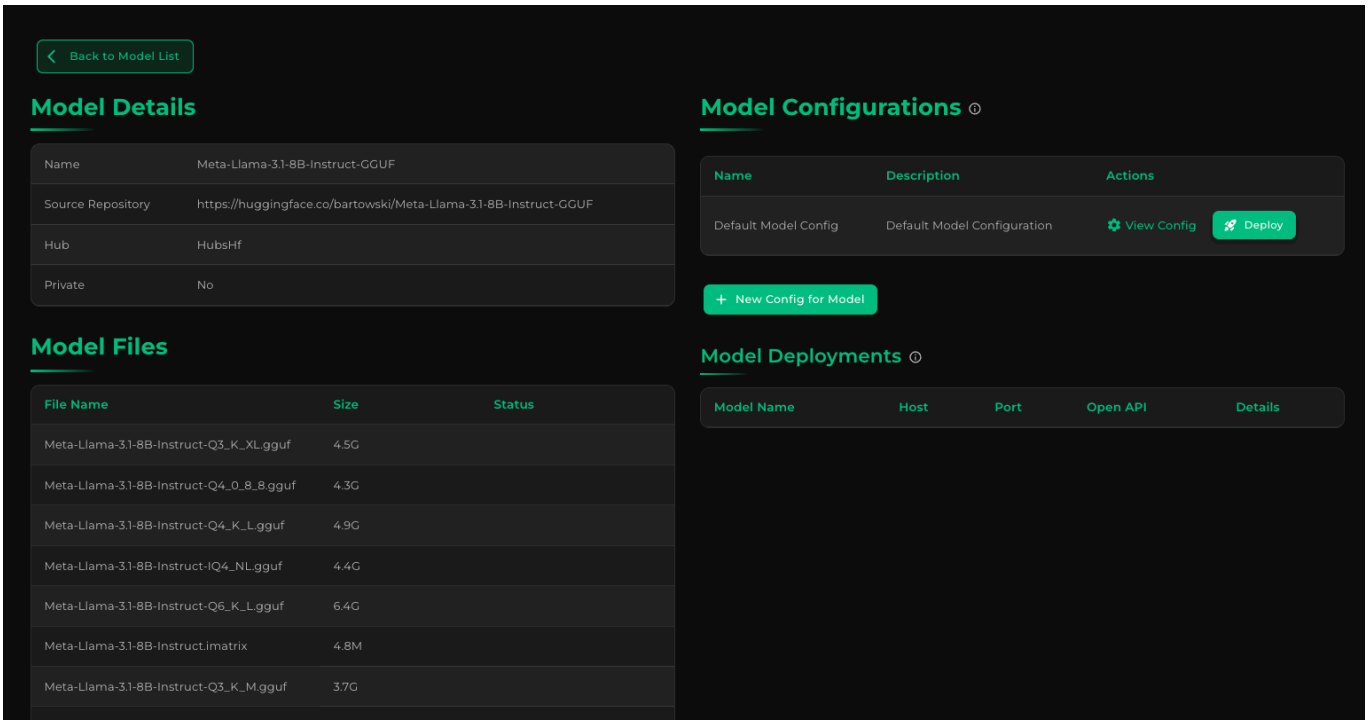
You will notice most of the files under **Model Files** have an empty status, since they were not downloaded. Scroll down to see if the 8-bit quantized GGUF is present, though:



As expected, the Q8_0-quantized GGUF is there. Since it is already marked as "downloaded", it is ready to deploy.

Step 5: Deploy a model for serving

The **Deploy** button appears at the upper right area of the Model Details screen, under Model Configurations:

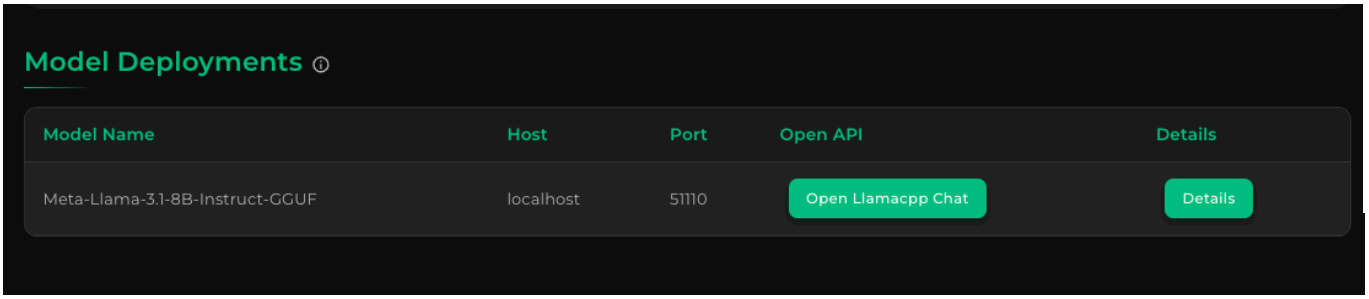


Kamiwaza creates a default config when a model is downloaded, to simplify the deployment experience.

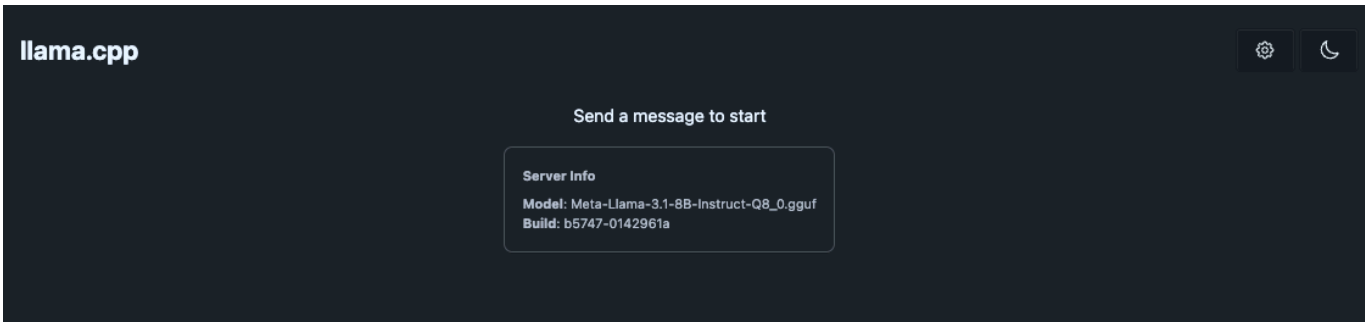
Click **"Deploy"** to launch an instance. When the spinner finishes, click **Back to Model List** to go back to the main Models screen.

Step 6: Testing the deployed model endpoint

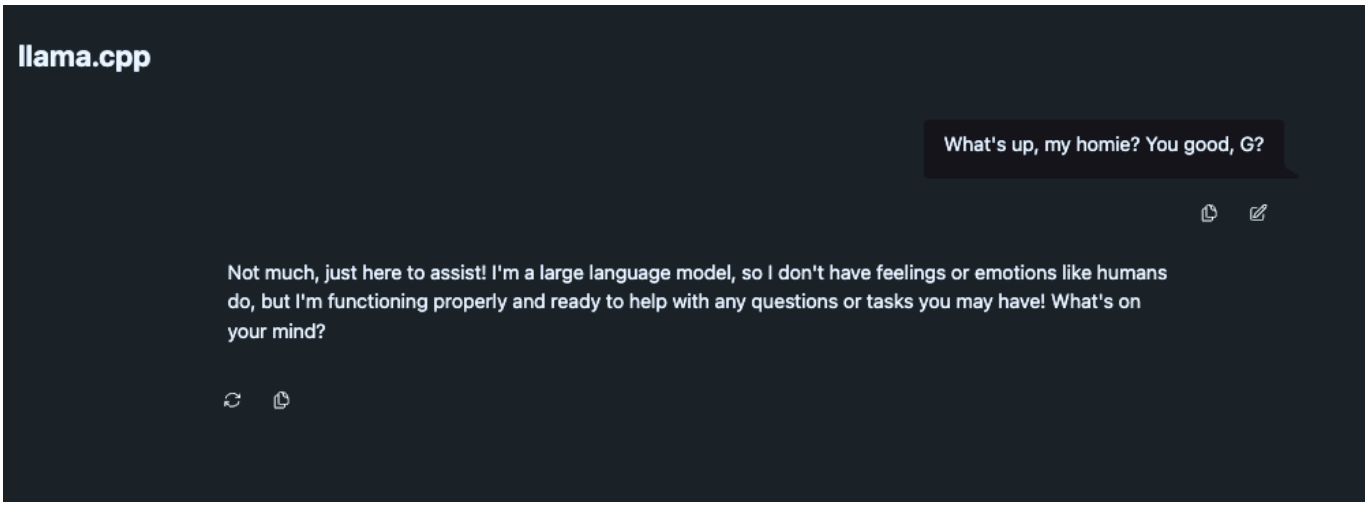
In the main Models screen, under Model Deployments, you will see a new entry for the recently deployed Llama 3.1 8B model:



To test the model, click "Open Llamacpp Chat". Your browser will open a chat application running against the deployed model:



Send a test message to validate that the LLM endpoint is working:



Perfect! If you see a response like that, then the inference endpoint is working as expected.

🎉 **You have successfully deployed your first model on Kamiwaza!** 🎉

Downloading Models

Model Search and Discovery

Kamiwaza is integrated directly with the Hugging Face Hub, allowing you to access a vast collection of open-source models. Models are identified by their Hugging Face repository ID, such as `meta-llama/Llama-3.3-70B-Instruct`.

Downloading Models

When you select a model to deploy, Kamiwaza handles the download process for you:

1. **Find the Model**: The system first checks if the model already exists locally. If not, it searches the Hugging Face Hub for the specified repository ID.
2. **File Selection**: For repositories containing multiple file types, you will want to select the most appropriate files for your hardware. For example, when running on hardware without a GPU, you can select and download specific GGUF files (`q6_k` , `q5_k_m` , etc.) for optimal performance with the `llama.cpp` engine. On a Linux server with GPUs, you may opt to download the standard model files (like `.safetensors`) for use with the `vLLM` engine.
3. **Local Caching**: All downloaded model files are stored in a local cache directory on the platform. This means that subsequent requests for the same model will not require a new download, making deployments faster.

Once a model's files are downloaded and verified, they are registered within Kamiwaza and become available for deployment.

Access control and hub behavior

- **ReBAC protection**: Model metadata and downloads are protected by relationship-based access control. The user who registers or downloads a model is automatically assigned ownership/viewer access.
- **Public viewer for hub models**: When a model is pulled from the Hugging Face Hub, Kamiwaza ensures a public viewer tuple exists so authorized users can discover the model without manual tuple edits.
- **Graceful hub rate limits**: If the Hugging Face Hub rate-limits requests, Kamiwaza returns partial results with an error indicator instead of failing the entire search.

Model Deployment

The model deployment process in Kamiwaza is designed to be simple and robust.

1. **Initiate Deployment**: When you request to deploy a model, Kamiwaza's `Models Service` takes over.
2. **Engine Selection**: The platform automatically determines the best engine based on your hardware, operating system, and the model's file format. For example, on a Mac with an M2 chip, a `.gguf` file will be deployed with `llama.cpp`, while `.safetensors` will use `MLX`. You can also override this and specify an engine manually.
3. **Resource Allocation**: The system allocates the compute resources needed for the selected runtime and configures the model's public route. Where the model lands — which node, which GPU, and how much memory is reserved — is decided by model placement; see the Model Placement Overview (see online documentation).
4. **Launch**: The selected engine is started in the environment appropriate for that deployment. In current Kubernetes deployments, this is typically a managed runtime workload exposed through the platform gateway.
5. **Health Check**: Kamiwaza monitors the model until it is healthy and ready to serve traffic.

Once deployed, your model is available via a standard API endpoint.

Deployment lifecycle statuses

Below are the deployment and instance statuses you may see, with what they mean and what (if anything) you should do.

- REQUESTED: The deployment request was accepted and recorded.
- DEPLOYING: Kamiwaza is creating the Ray Serve app (if applicable) and preparing routing.
- INITIALIZING: Routing is up and the model server is reachable, but the model is still loading or not yet ready. Normal for a short period right after launch.
- DEPLOYED: The deployment is healthy and ready to serve traffic.
- STOPPED: The deployment was stopped (either by a user action or system shutdown).
- ERROR: A recoverable problem was detected. Often resolves after a change or retry. See error code guidance below.
- FAILED: A terminal failure was detected (e.g., out-of-memory). Requires user action to resolve.
- MUST_REDOWNLOAD: Required weights are missing locally in community installs. Re-download the model and deploy again.

Instance-level statuses (for replicas):

- REQUESTED: An instance record was created and is queued to start.
- COPYING_FILES: Required files are being synced to the node.
- DEPLOYED (instance): The process is launching or up and responding.

Error codes and what to do

If a deployment shows ERROR or FAILED, the UI may show a short error code and message. Common codes:

- OOM (Out of Memory): Reduce context size, select a smaller model/variant, or lower GPU memory utilization.
- CUDA_ERROR: Check GPU drivers/availability; restart GPU services or the host if needed; ensure the container has GPU access.
- MODEL_LOADING_FAILURE: Verify that all model files exist, are accessible, and match the expected version; try re-downloading.
- CONTAINER_EXITED: The runtime process crashed. Open logs for details; check memory limits, incompatible flags, or driver issues.
- RUNTIME_ERROR: A generic runtime exception was seen in logs. Open logs for specifics.
- STARTUP_TIMEOUT: The model did not become ready within the expected time. Try a smaller model/context or adjust engine parameters.
- MUST_REDOWNLOAD: Files missing locally (community installs). Re-download the model and retry.

Viewing logs and diagnostics

- In the advanced UI, open a deployment row and click “View logs” to see container logs and auto-detected issue patterns (OOM, CUDA errors, etc.).

Tips for Novice mode

- If you hit OOM or STARTUP_TIMEOUT, try:
  - Selecting a smaller preset (model/variant)
  - Reducing context size (the UI will suggest balanced options)
  - Re-deploying after downloads complete

When to retry vs. change configuration

- Retry directly if you see transient ERROR without an error code.
- Change configuration if you see a clear code like OOM, MODEL_LOADING_FAILURE, CONTAINER_EXITED, or STARTUP_TIMEOUT.

How routing works

Current Kamiwaza deployments use path-based routing as the standard public access model. Routes can be created immediately after launch while the runtime continues its readiness checks. This is why you may see `INITIALIZING` briefly before `DEPLOYED`.

Deployments are exposed on a single host using path prefixes. OpenAI-compatible base URL:

```
https://<host>/runtime/models/<deployment_id>/v1
```

Some older environments may still expose legacy port-based routes, but path-based URLs are the canonical pattern for current customer-facing documentation. See Routing & URLs (see online documentation) for details.

Tenant-mode inference

> **SUPPORT SCOPE**
>
> This page documents tenant-mode inference for Kamiwaza 1.3.4. Support covers the configurations marked Supported in Compatibility scope (see online documentation), qualified on k0s Kubernetes 1.36.

Tenant-mode inference is how a namespaced Kamiwaza install serves local models on GPUs and CPUs. Kamiwaza installs into one namespace and cannot read the cluster's Nodes, so it cannot discover GPUs itself. The cluster owner describes the hardware instead, in a profile that the owner signs, and Kamiwaza places each deployment on hardware the signed profile vouches for.

A default install has tenant-mode inference off and cannot use GPUs: a GPU deployment is refused with HTTP 422. To serve models on GPUs:

1. Prepare the GPU nodes; see GPU Nodes (see online documentation).
2. Create the owner key, detect each GPU host's facts, and sign a profile, with the owner tool in the Kamiwaza core image. See Turn on tenant-mode inference (see online documentation).
3. Point the install at the signed profile, in four Helm values.

Compatibility scope

Live qualification for this release targets **k0s Kubernetes 1.36 only**. Chart, API, and client contract checks cover Kubernetes 1.28 through 1.36, but passing a contract check is not a live support claim for that minor version: only 1.36 has been exercised end to end. The allocation classes are:

Class	Engines	Kubernetes resource	Isolation statement	Release status
CPU	llama.cpp, whisper.cpp	`cpu` and memory requests	Kubernetes scheduler allocation	Supported on k0s 1.36
Whole NVIDIA GPU	llama.cpp, vLLM, diffusion (CUDA)	`nvidia.com/gpu`	Whole-device allocation	Supported on k0s 1.36, one device per replica only. A deployment that spans more than one device is implemented but not qualified live
Whole AMD GPU	None in this release; no ROCm engine image ships	`amd.com/gpu`	Whole-device allocation	**Not qualified in this release.** Do not deploy against a support claim
NVIDIA MIG	llama.cpp, vLLM, diffusion (CUDA)	Qualified `nvidia.com/mig-*`	Hardware partition, exact shape only	Supported on k0s 1.36, recorded slice shape only
VRAM-plugin sharing	llama.cpp, vLLM, diffusion (CUDA)	Owner-advertised `kamiwaza.ai/vram-gb-*`	Accounted sharing, not hard isolation	Supported on k0s 1.36, recorded plugin and profile only. Optional: the cluster owner installs the VRAM plugin (see online documentation)

Engines

Tenant mode serves only the engines in the platform engine catalog that ships inside the release's core image:

Engine	Serves	Placement	Images
llama.cpp	Text generation; embeddings when the model configuration sets `embedding` and states its `pooling`	GPU or CPU	NVIDIA CUDA 13 and CUDA 12, amd64 and arm64; CPU amd64 and arm64
vLLM	Text generation	GPU only	NVIDIA CUDA 13, amd64 and arm64 (GB10 only)
Diffusion	Image generation	GPU only, one device per replica	NVIDIA CUDA 12 (built with 12.8), amd64 only
whisper.cpp	Transcription	CPU only	CPU amd64 and arm64

The engine catalog has no arm64 image for diffusion in this release. A binding on an arm64 node therefore cannot serve diffusion, and the platform refuses the deployment with `no diffusion image is published for nvidia/arm64`.

vLLM's arm64 image serves one card, the GB10 (DGX Spark, compute capability 12.1). The platform refuses vLLM on any other arm64 NVIDIA card, such as GH200, GB200, Jetson Thor, or a discrete Blackwell card in an arm64 server, with a reason such as `vllm requires compute capability 12.1; this binding declares 10.0`.

A GPU binding serves an engine only when the engine publishes an image for the binding's vendor and architecture, and the hardware its `accelerator` block attests meets the engine's requirements. Each NVIDIA image is built against one CUDA major and carries kernels down to a lowest compute capability, so the card and the host driver both limit which image can run:

Engine	NVIDIA image	Architectures	Lowest compute capability	Host driver
llama.cpp	CUDA 13	amd64, arm64	7.5 (Turing)	580.65.06 (R580) or later
llama.cpp	CUDA 12	amd64, arm64	7.0 (Volta)	525.60.13 or later; R570 or later for Blackwell GPUs
vLLM	CUDA 13	amd64	7.5, and the engine requires `bf16` (8.0 or later)	580.65.06 (R580) or later
vLLM	CUDA 13	arm64	12.1 (GB10 only)	580.65.06 (R580) or later
Diffusion	CUDA 12 (built with 12.8)	amd64	8.0, and the engine requires `bf16`	R570 or later recommended, not enforced

llama.cpp therefore requires compute capability 7.0 or later: a Volta card runs the CUDA 12 image, and no image serves Pascal or older. vLLM requires a CUDA 13 driver. For how the platform picks between the llama.cpp images, see Attest the NVIDIA driver's CUDA major (see online documentation).

The platform compares only the driver's CUDA major, so it does not refuse the diffusion image on an older CUDA 12 driver. The image is built with PyTorch for CUDA 12.8. Unlike the llama.cpp CUDA 12 image, which runs on older CUDA 12 drivers under NVIDIA's minor-version compatibility, the diffusion image is not qualified below R570, so run diffusion on R570 or later.

A diffusion deployment serves one diffusers pipeline on one device through `/v1/images/generations`. The model must be a diffusers repository with `model_index.json` at its top; a single-file checkpoint is refused. A pipeline has no context length, so the platform sizes the device from the pipeline's weights. Tenant mode refuses a diffusion deployment that configures LoRA or Lightning adapters, a stream profile, `trust_remote_code`, DFloat11 settings, or more than one device (`tensor_parallel_size` or `gpu_count` above 1). When a configuration key causes the refusal, the reason names that key. A DFloat11 model that the platform recognizes by its name is also refused; that reason says DFloat11 models need more than one device and names no key. A stream profile that the platform detects from the model's name is kept, not refused.

When the release sets `core.inferenceResources.enabled: true`, every local deployment takes the tenant path, including a request without an `inferenceResources` block, which the platform derives. There is no fallback to the standard deployment path. An engine that the engine catalog does not list is refused. MLX is not in the catalog, so MLX is not supported in tenant mode in this release. A deployment without a block that selects such an engine receives HTTP 422, and the `detail` ends with the engine and the reason:

```
cannot derive inference resources for this deployment: engine 'mlx' is not qualified in
this release
```

A request that carries its own block is accepted and then fails at dispatch with the same reason.

External endpoints (see online documentation) are not refused. An external endpoint proxies a remote service and allocates no local resources, so on a tenant cluster the platform does not size it and creates no workload in the tenant namespace. On a tenant cluster, two external-endpoint requests receive HTTP 422:

- A request that supplies an `inferenceResources` block:

  ```
  external endpoints allocate no local resources; omit inferenceResources
  ```

- A request that sets a nonzero `gpu_allocation` or `vram_allocation`. A cluster without tenant resources still honors these overrides:

  ```
  external endpoints allocate no local resources; omit gpu_allocation and
  vram_allocation
  ```

One replica across several devices

A deployment can span up to eight whole devices in a single Pod through tensor parallelism. This is implemented but not qualified live in this release; see Compatibility scope (see online documentation). The device count comes from `tensor_parallel_size` in the model configuration, or from a signed recipe when one pins it. An absent value is one device — never the node's device count, so a first deployment does not take a whole multi-GPU node.

Tensor parallelism needs a peer-to-peer path between the devices, so it is admitted only on a whole-device binding whose `accelerator` block attests the `p2p` feature. The owner tool's detector claims `p2p` only when every pair of whole-mode GPUs on the host reports a peer path, because the device plugin can give a multi-GPU request any whole GPUs. MIG-mode GPUs are left out: a whole-device binding is never given one. On a host whose GPUs are bridged in pairs, partition the GPUs outside one bridged pair with MIG, so its whole GPUs are exactly that pair; see MIG and tensor parallelism (see online documentation).

A MIG partition does not have a peer path, and an accounted VRAM share is a slice of a device the workload does not exclusively hold, so both refuse a count above one rather than quietly serving a single device to an engine told to shard across several. A count that is not a whole number, is below one, or exceeds eight is refused for the same reason: the request and the Pod must agree.

Pipeline and data parallelism are not part of this release.

Ownership boundary

The cluster owner supplies drivers, device plugins, optional VRAM sharing, RuntimeClasses, admission policy, the signed profile, and the `kamiwaza-owner-accelerators` quota that caps the accelerators any pod in the namespace can request (see Cap the namespace's accelerators (see online documentation)). Every GPU binding in a profile attests its hardware in an `accelerator` block, which the owner tool detects on the host; a CPU

binding needs only its architecture. Recipe catalogs are optional, GPU and CPU alike: without one, the platform derives each deployment, and a published recipe pins a specific model and configuration. The tenant supplies only namespaced Helm and serving inputs.

Kubernetes namespace-admin permissions do not grant the Kamiwaza `admin` role. Model deployment through the API, SDK, or UI remains Kamiwaza-admin-only.

A deployment request needs no `inferenceResources` block: the platform derives one. For a GPU deployment it sizes memory from the model's own metadata and the context the model configuration sets (`max_model_len`), or else the context the model declares, and places the request on the smallest binding that holds it. A deployment goes to CPU instead when it sets `force_cpu`, when its engine ships only a CPU build (whisper.cpp), or when the owner declared no GPU hardware; it is sized from the platform's CPU floor and the model's weights, and runs the platform's CPU build for the binding's architecture. A request that cannot be served as asked -- no context declared anywhere, or an embedding configuration that does not state its `pooling` -- is refused with HTTP 422 and the reason; one the platform cannot evaluate, such as an unreadable profile bundle, with HTTP 503.

The tenant chart creates no cluster-scoped GPU resource and does not require `nodes/get`, `nodes/list`, or `nodes/watch`. A profile or allocator failure is reported as a failure; the platform does not silently fall back to CPU or a weaker isolation class.

Turn on tenant-mode inference

Before you start

You need	Notes
Prepared GPU nodes	Every requirement on GPU Nodes (see online documentation) passes. Skip for CPU-only serving.
The release's core image, by digest	The owner tool ships in it. Sign with the core image of the release you install, so the release that verifies the profile is the one that signed it.
Docker or Podman	On the machine where you sign, not on the GPU nodes: step 3 runs the detector as a pod. The owner tool runs in the core image; it contacts no cluster and no network.
`kubectl` and `jq`	On the machine where you publish the profile.

> **DO NOT INSTALL DOCKER ON A KUBERNETES NODE**
>
> Docker sets the host's iptables `FORWARD` policy to `DROP`, which breaks pod networking on the node. Run the owner tool on a separate machine, and the detector as a pod.

Read the core image from a running install:

```
kubectl -n kamiwaza get deploy core-scheduler \
  -o jsonpath='{.spec.template.spec.containers[?(@.name=="core")].image}'; echo
```

```
<YOUR_REGISTRY>/kamiwaza-ai/releases/kamiwaza/images/core@sha256:<DIGEST>
```

Before the install, take the same reference from the release's image list; see Install Without Internet Access (see online documentation).

If your registry requires credentials, log the container engine in to it first: `docker login <REGISTRY_HOST>`, or `podman login <REGISTRY_HOST>`. Podman also reads an existing credentials file named with `--authfile <FILE>` or the `REGISTRY_AUTH_FILE` environment variable, in the same format as the install's pull Secret.

Set the image, and define a shell function that runs the owner tool with the current directory mounted. With Docker:

```
export CORE_IMAGE="<YOUR_REGISTRY>/kamiwaza-ai/releases/kamiwaza/images/core@sha256:
<DIGEST>"
owner_tool() {
  docker run --rm -i --user "$(id -u):$(id -g)" -v "$PWD:/work" -w /work \
    "$CORE_IMAGE" python -m kamiwaza.serving.inference_resources.owner_tool "$@"
}
```

With rootless Podman, which needs `--userns=keep-id` to write files you own, and `:Z` on the mount to use the directory on a host with SELinux, such as RHEL or Fedora:

```
export CORE_IMAGE="<YOUR_REGISTRY>/kamiwaza-ai/releases/kamiwaza/images/core@sha256:
<DIGEST>"
owner_tool() {
  podman run --rm -i --userns=keep-id --user "$(id -u):$(id -g)" -v "$PWD:/work:Z" -w
/work \
    "$CORE_IMAGE" python -m kamiwaza.serving.inference_resources.owner_tool "$@"
}
```

`:Z` relabels the current directory for the container, and Podman ignores it on a host without SELinux. Podman refuses to relabel your home directory, so run the function from a directory of its own.

The steps below use the function; the GPU detector in step 3 runs on the GPU node instead, because it must see the GPUs.

Step 1: Create the owner signing key

```
owner_tool keygen --private-key-out owner-ed25519-private.pem
```

```
{
  "keyId": "sha256:6e1f...",
  "publicKey": "8Ljv..."
}
```

The tool writes an unencrypted Ed25519 private key readable only by you, and refuses to replace an existing file. It prints the public key, which goes into the Helm values as `ownerPublicKey`, and the `keyId` your profile names. Keep the private key out of source control, out of the tenant namespace, and out of any evidence you record. To print the public half again:

```
owner_tool pubkey --private-key owner-ed25519-private.pem
```

Step 2: Choose a cluster binding ID

Pick a `clusterBindingId` for this cluster: 1 to 128 characters, starting with a letter or digit, then letters, digits, `.`, `_`, `:`, or `-`. The profile and the Helm values must carry the same value; Kamiwaza refuses a profile signed for another binding.

Step 3: Detect each GPU host's facts

Run the detector on every GPU node, with all of the node's GPUs visible to it. Run it as a one-shot privileged pod on the `nvidia` RuntimeClass: the node then needs no container engine of its own. It must be privileged because GPU nodes give an unprivileged pod only the GPUs a device plugin allocated to it (see GPU Nodes (see online documentation)). So it runs in a namespace of its own, never the Kamiwaza namespace, where no pod may be privileged.

Create that namespace once, and copy the install's pull Secret into it so the pod can pull the core image. If your install has no pull Secret, skip the copy and delete the two `imagePullSecrets` lines below:

```
kubectl create namespace kamiwaza-owner-detect
kubectl label namespace kamiwaza-owner-detect --overwrite \
  pod-security.kubernetes.io/enforce=privileged
kubectl -n kamiwaza get secret kamiwaza-pull -o json \
  | jq '{apiVersion, kind, type, data, metadata: {name: .metadata.name}}' \
  | kubectl -n kamiwaza-owner-detect apply -f -
```

Then, for each GPU node, set `NODE_NAME` to the node and run:

```
NODE_NAME="<NODE_NAME>"
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: owner-tool-detect
  namespace: kamiwaza-owner-detect
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  runtimeClassName: nvidia
  restartPolicy: Never
  nodeSelector:
    kubernetes.io/hostname: $NODE_NAME
  imagePullSecrets:
  - name: kamiwaza-pull
  containers:
  - name: detect
    image: $CORE_IMAGE
    command: ["python", "-m", "kamiwaza.serving.inference_resources.owner_tool",
"detect"]
    securityContext:
      privileged: true
    env:
    - {name: NVIDIA_VISIBLE_DEVICES, value: all}
    - {name: NVIDIA_DRIVER_CAPABILITIES, value: "compute,utility"}
EOF
kubectl -n kamiwaza-owner-detect wait --for=jsonpath='{.status.phase}'=Succeeded
pod/owner-tool-detect --timeout=5m
kubectl -n kamiwaza-owner-detect logs owner-tool-detect
kubectl -n kamiwaza-owner-detect delete pod owner-tool-detect
```

When every node is done, delete the namespace, which also deletes its copy of the pull Secret:

```
kubectl delete namespace kamiwaza-owner-detect
```

The detector prints the `accelerator` block for the node's GPUs:

```
{
  "accelerator": {
    "computeCapability": "8.0",
    "cudaDriverMajor": 13,
    "features": [
      "bf16",
      "mig",
      "nvlink",
      "p2p"
    ],
    "vendor": "nvidia"
  }
}
```

A GPU node that already has a container engine can run the detector directly instead, privileged for the same reason as the pod. With Podman, as root, through the NVIDIA runtime; add `--authfile <FILE>` if your registry requires credentials:

```
sudo podman run --rm --privileged --runtime /usr/bin/nvidia-container-runtime \
  -e NVIDIA_VISIBLE_DEVICES=all "$CORE_IMAGE" \
  python -m kamiwaza.serving.inference_resources.owner_tool detect
```

`podman run --device nvidia.com/gpu=all` does not work with Podman 4.9, the version Ubuntu 24.04 ships: it cannot read the device description that current NVIDIA container toolkits write, and fails with `unresolvable CDI devices nvidia.com/gpu=all`. On a node that already runs Docker with the NVIDIA runtime, `docker run --rm --privileged --gpus all "$CORE_IMAGE" python -m kamiwaza.serving.inference_resources.owner_tool detect` also works. Do not install Docker on a Kubernetes node to run it.

The detector reads the facts from `nvidia-smi` instead of expecting you to know them:

- `computeCapability` is the GPUs' compute capability. The detector refuses a node with two kinds of GPU: a binding cannot single out one kind on such a node, so keep it out of tenant-mode GPU serving; see GPU Nodes (see online documentation).
- `cudaDriverMajor` is the major of the `CUDA Version` in `nvidia-smi`'s header. The detector exits with an error rather than guess when it cannot read it.
- `bf16` and `fp8` follow from the compute capability, and `nvlink` from the links the GPUs report.
- `p2p` appears only when every pair of whole-mode GPUs has a peer path; see One replica across several devices (see online documentation).
- `mig` appears when the GPUs can be partitioned, whether or not a GPU is in MIG mode now. It is informational: no engine requires it, and a MIG binding does not need it.

Copy the block into the binding for those nodes. A binding covers every node of its architecture that advertises its resource (see `nodeSelector` in step 4), so when those nodes differ, attest only what holds on all of them: the lowest compute capability and `cudaDriverMajor`, and only the features every node has. Drop `p2p` and `nvlink` from a MIG or VRAM-share binding: a slice has neither.

Step 4: Write the profile envelope

The envelope lists the profiles the cluster offers. Each profile has one guarantee and one or more bindings; each binding names the Kubernetes resource a deployment requests and the nodes it runs on. This envelope offers CPU serving on amd64 nodes and whole NVIDIA GPUs. Save it as `owner-profile-envelope.json`:

```
{
  "schemaVersion": 1,
  "clusterBindingId": "<CLUSTER_BINDING_ID>",
  "namespace": "kamiwaza",
  "version": 1,
  "keyId": "<KEY_ID>",
  "issuedAt": "<ISSUED_AT>",
  "expiresAt": "<EXPIRES_AT>",
  "evidenceId": "gpu-nodes-checked-2026-10-01",
  "profiles": [
    {
      "name": "cpu-standard",
      "capability": "cpu",
      "guarantees": [],
      "compatibleVariantIds": ["cpu-default"],
      "bindings": [{
        "id": "cpu-amd64",
        "allocator": "cpu",
        "architecture": "amd64",
        "runtime": {"kind": "cpu", "variantId": "cpu-default", "image": "
<LLAMACPP_CPU_IMAGE>"},
        "capacity": {"kind": "scheduler"},
        "nodeSelector": {"kubernetes.io/arch": "amd64"}
      }]
    },
    {
      "name": "nvidia-whole",
      "capability": "gpu",
      "guarantees": ["whole-device"],
      "compatibleVariantIds": ["llamacpp-nvidia-amd64", "vllm-nvidia-amd64",
"diffusion-nvidia-amd64"],
      "bindings": [{
        "id": "nvidia-whole-amd64",
        "allocator": "extended-resource",
        "architecture": "amd64",
        "accelerator": {
          "vendor": "nvidia",
          "computeCapability": "8.0",
          "cudaDriverMajor": 13,
          "features": ["bf16", "mig", "nvlink", "p2p"]
        },
        "capacity": {"kind": "scheduler", "usableMemory": "79Gi"},
        "resource": {"name": "nvidia.com/gpu", "quantity": 1},
        "runtime": {"kind": "cuda", "variantId": "llamacpp-nvidia-amd64", "image": "
<LLAMACPP_CUDA_IMAGE>"},
        "runtimeClassName": "nvidia",
        "nodeSelector": {"kubernetes.io/arch": "amd64"}
      }]
    }
  ]
}
```

Fill in the placeholders:

Field	Value
`clusterBindingId`	The ID from step 2.
`namespace`	The namespace Kamiwaza is installed in.
`version`	An integer you raise with every new revision. Kamiwaza refuses a revision below `core.inferenceResources.minimumBundleVersion` (default `1`).
`keyId`	The `keyId` from step 1.
`issuedAt`, `expiresAt`	UTC timestamps with a `Z` suffix, such as `2026-10-01T00:00:00Z`. Kamiwaza refuses the profile outside that window, so publish a new revision before `expiresAt`.
`evidenceId`	A label for the checks behind this revision, for your records.
`accelerator`	The block from step 3.
`nodeSelector`	`kubernetes.io/arch` only; this release accepts no other key. A binding therefore covers every node of that architecture that advertises its resource, so its `accelerator` facts and `usableMemory` must hold on all of them.
`usableMemory`	The GPU memory a deployment may plan on, less what the driver and CUDA context hold back. Write a whole number of `Gi`, `Mi` or `Ki`; the profile schema refuses a fraction such as `121.7Gi`. The right value depends on the binding; see the list below this table.
`runtime.image`	A digest-pinned engine image. Derived deployments take their image from the engine catalog; this value is used only for a deployment that a recipe pins, which runs it as written. In a mirrored or air-gapped install, name your mirror's copy. Print the catalog's images with the command below.

Set `usableMemory` for each kind of binding as follows:

- **Whole device or MIG partition on a discrete card:** the device's or the slice's memory less the driver's and CUDA context's share, `79Gi` on an 80 GB card. Understating it is the safe direction.
- **VRAM plugin:** state it accurately. vLLM reserves a fraction of the whole card: 90%, the engine catalog (see online documentation) default, times the request divided by `usableMemory`. Understating `usableMemory` therefore raises what vLLM reserves, and the reservation passes the grant once `usableMemory` is about a tenth below the card's memory.
- **Whole-device GB10:** its GPU memory is the host's, so set `usableMemory` to the device total rounded down, `121Gi` on a DGX Spark, whose total is about 121.69 GiB. vLLM's budget is the request divided by `usableMemory`, rounded up to a hundredth, applied to that total. At `121Gi` vLLM already reserves slightly more than the request: 121.69/121 of it, plus up to 0.01 of the total from the rounding. A smaller value reserves further above the request. The host's own processes use the same memory, so a request must also fit in what the host leaves free. A VRAM-plugin binding on a GB10 follows the VRAM-plugin rule instead.

A GPU profile's `compatibleVariantIds` must list `<engine>-nvidia-<architecture>` for each engine it may serve, and each binding's `runtime.variantId` must be in its profile's list. To print the engine images the core image carries:

```
docker run --rm "$CORE_IMAGE" python -c \
  "from importlib.resources import files;
print(files('kamiwaza.serving.inference_resources').joinpath('engine-
catalog.json').read_text())" \
  | jq -r '.engines[] | .engine as $e | .images[] | [$e, .vendor, .architecture, (.cuda
// "-" | tostring), .image] | @tsv'
```

```
llamacpp        nvidia  amd64   13      ghcr.io/kamiwaza-
internal/containers/images/llamacpp-cuda13@sha256:a482...
llamacpp        cpu     amd64   -       ghcr.io/kamiwaza-
internal/containers/images/llamacpp-cpu@sha256:ccbd...
```

For MIG, add a profile whose guarantee is `mig` and whose binding requests the partition the node advertises. This profile offers the `3g.40gb` slices from GPU Nodes (see online documentation); add it to the `profiles` list:

```
{
  "name": "nvidia-mig-3g40gb",
  "capability": "gpu",
  "guarantees": ["mig"],
  "compatibleVariantIds": ["llamacpp-nvidia-amd64", "vllm-nvidia-amd64", "diffusion-
nvidia-amd64"],
  "bindings": [{
    "id": "nvidia-mig-3g40gb-amd64",
    "allocator": "extended-resource",
    "architecture": "amd64",
    "accelerator": {
      "vendor": "nvidia",
      "computeCapability": "8.0",
      "cudaDriverMajor": 13,
      "features": ["bf16", "mig"]
    },
    "capacity": {"kind": "scheduler", "usableMemory": "38Gi"},
    "resource": {"name": "nvidia.com/mig-3g.40gb", "quantity": 1},
    "runtime": {"kind": "cuda", "variantId": "llamacpp-nvidia-amd64", "image": "
<LLAMACPP_CUDA_IMAGE>"},
    "runtimeClassName": "nvidia",
    "nodeSelector": {"kubernetes.io/arch": "amd64"}
  }]
}
```

The `accelerator` block is the node's, from step 3, without `p2p` and `nvlink`: a slice has neither. Set `usableMemory` to the slice's memory less headroom for the CUDA context. On the verification host that GPU Nodes (see online documentation) describes, a `3g.40gb` slice of an A100 80GB reports 40192 MiB in `nvidia-smi`, so the example uses `38Gi`. Binding `id` values must be unique across the whole envelope.

An `accounted-share` profile, for the VRAM plugin (see online documentation), uses the `vram-plugin` allocator and `{"kind": "accounted", ...}` capacity, and reserves a bookkeeping bucket; it does not provide hard VRAM or compute isolation.

Step 5: Sign and publish the profile

```
owner_tool sign --envelope owner-profile-envelope.json \
  --private-key owner-ed25519-private.pem > owner-profile-configmap.json
```

The tool runs Kamiwaza's own verifier over what it signs, so it refuses a profile this release would reject, and names the check that failed:

```
error: signed bundle would be refused: invalid inference profile authorization: bundle
is expired or its window is empty
```

It emits an immutable ConfigMap named `kamiwaza-inference-profiles`. Give each revision its own name, then apply it:

```
export PROFILE_CONFIGMAP=kamiwaza-inference-profiles-r1
jq --arg name "$PROFILE_CONFIGMAP" '.metadata.name = $name' owner-profile-
configmap.json \
  > "$PROFILE_CONFIGMAP.json"
kubectl apply --server-side --field-manager=owner-inference-publisher -f
"$PROFILE_CONFIGMAP.json"
```

Check:

```
kubectl -n kamiwaza get configmap "$PROFILE_CONFIGMAP" -o jsonpath='{.immutable}
{.data.bundle\.sha256}'; echo
```

```
true 9c1d...
```

Anchor the profile on the namespace

As the cluster owner, anchor the profile you published with one annotation on the `kamiwaza` namespace. Kamiwaza then trusts only the key, cluster binding, and profile ConfigMap the annotation names, and refuses a profile whose `version` is lower than the anchored one, whatever the Helm values say. The namespace administrator can read the annotation but cannot change or remove it, so a superseded profile cannot be put back.

The owner tool writes the annotation as a patch file from the published ConfigMap. It sets the minimum version to this profile's `version`, and checks that Kamiwaza would accept the profile under it. It needs no private key. Run the block as one command, so the patch is applied only when the tool succeeds:

```
owner_tool anchor --configmap "$PROFILE_CONFIGMAP.json" \
  --public-key <OWNER_PUBLIC_KEY> > owner-anchor-patch.json \
  && kubectl patch namespace kamiwaza --type merge --patch-file owner-anchor-patch.json
```

If the tool refuses the profile, nothing is applied and the namespace keeps its current anchor. `kubectl patch` also refuses an empty or missing patch file. Do not set the annotation from the patch file any other way, such as with `kubectl annotate`: if the tool failed, that writes an empty annotation, which is malformed and refuses every deployment.

Check:

```
kubectl get namespace kamiwaza \
  -o jsonpath='{.metadata.annotations.inference\.kamiwaza\.ai/owner-anchor}'; echo
```

```
{"bundleConfigMap":"kamiwaza-inference-profiles-
r1","clusterBindingId":"...","minimumBundleVersion":1,"publicKeys":
["8Ljv..."],"schemaVersion":1}
```

The anchor is optional in this release and is planned to be required from 1.4. Without it, Kamiwaza takes its trust from the Helm values in step 6 alone and logs `UNANCHORED OWNER TRUST`, and anyone who can change those values can point Kamiwaza at an older profile or another key. A malformed anchor refuses every deployment rather than falling back to the Helm values.

Cap the namespace's accelerators

Any identity that can create pods in the `kamiwaza` namespace, including the namespace administrator and Kamiwaza's own service accounts, can request any accelerator a node advertises, outside the profile: a whole GPU the profile shares only through VRAM accounting, a MIG slice it never granted, or all of a VRAM bucket. As the cluster owner, cap that with one ResourceQuota, `kamiwaza-owner-accelerators`. Kubernetes enforces it and Kamiwaza never reads it. The namespace administrator can read it but cannot change or remove it.

The quota is optional in this release, like the anchor, and is planned to be required from 1.4.

The owner tool renders the quota from the published profile and a snapshot of your nodes. It checks the profile as `owner_tool anchor` does, and contacts no cluster:

- Each accelerator a GPU binding names is set to what the nodes the binding selects advertise. A VRAM bucket counts at most its `totalUnits` from each node that advertises it, and the quota is the sum: it caps the namespace's total across those nodes, not each node. With a single node, the quota is `totalUnits`.
- Every other NVIDIA, AMD, Intel or VRAM-bucket resource is set to `0`.
- Every key is `requests.<resource>`, the only form a quota counts for these resources. Kubernetes accepts `limits.` and bare keys but counts nothing against them, so do not write the quota by hand.

As the cluster owner, with cluster-admin credentials, take the node snapshot, then render and apply the quota as one command, so nothing is applied when the tool refuses the profile:

```
kubectl get nodes -o json > nodes.json
owner_tool quota --configmap "$PROFILE_CONFIGMAP.json" \
  --public-key <OWNER_PUBLIC_KEY> --allocatable nodes.json > owner-quota.json \
  && kubectl apply --server-side --field-manager=owner-inference-publisher -f owner-
quota.json
```

`--limit <RESOURCE>=<COUNT>` lowers the count of a resource the profile grants. It cannot grant a resource the profile does not, or raise a VRAM bucket above its `totalUnits`.

If the `kamiwaza-extensions` namespace exists and no extension needs an accelerator, cap it at zero. The quota cannot grant an accelerator the profile does not, and that namespace has no profile, so a zero quota refuses every accelerator an extension asks for. First check that no pod there requests one; the command prints nothing when none does:

```
kubectl -n kamiwaza-extensions get pods -o json | jq -r '.items[]
  | select([.spec.containers[], (.spec.initContainers // [])[]
      | .resources.requests // {} | keys[]
      | select(test("^(nvidia\\.com|amd\\.com|gpu\\.intel\\.com)/|^kamiwaza\\.ai/vram-
"))]
    | length > 0)
  | .metadata.name'
```

Then render and apply it. Without `--configmap`, every accelerator is `0`:

```
owner_tool quota --allocatable nodes.json --namespace kamiwaza-extensions \
  > extensions-quota.json \
  && kubectl apply --server-side --field-manager=owner-inference-publisher -f
extensions-quota.json
```

Check: the quota lists its limits and current use, and the namespace administrator is refused a pod that asks for an accelerator the profile does not grant:

```
kubectl -n kamiwaza get resourcequota kamiwaza-owner-accelerators \
  -o jsonpath='{.status.hard}{"\n"}{.status.used}{"\n"}'
kubectl -n kamiwaza run quota-probe --image=busybox:1.37 --restart=Never --dry-
run=server \
  --annotations=sidecar.istio.io/inject=false \
  --overrides='{"spec":{"securityContext":
{"runAsNonRoot":true,"runAsUser":65532,"seccompProfile":
{"type":"RuntimeDefault"}},"containers":[{"name":"quota-
probe","image":"busybox:1.37","securityContext":
{"allowPrivilegeEscalation":false,"capabilities":{"drop":["ALL"]}},"resources":
{"requests":{"amd.com/gpu":"1"},"limits":{"amd.com/gpu":"1"}}}]}}' \
  --as=system:serviceaccount:kamiwaza:kamiwaza-installer \
  --as-group=system:serviceaccounts --as-group=system:serviceaccounts:kamiwaza \
  --as-group=system:authenticated
```

```
{"requests.amd.com/gpu":"0","requests.nvidia.com/gpu":"2",...}
{"requests.amd.com/gpu":"0","requests.nvidia.com/gpu":"0",...}
Error from server (Forbidden): pods "quota-probe" is forbidden: exceeded quota:
kamiwaza-owner-accelerators, requested: requests.amd.com/gpu=1, used:
requests.amd.com/gpu=0, limited: requests.amd.com/gpu=0
```

The probe pod meets the restricted Pod Security standard, so it reaches the quota check even in a namespace that enforces that standard. Use above a limit means pods admitted before the quota still hold those accelerators. They keep running; find out why.

Once the quota is in place:

- A deployment over the quota is refused when its pod is created, not left pending, and stays pending with the reason `exceeded quota: kamiwaza-owner-accelerators`. Kubernetes retries it with a backoff that doubles, about 3, 5.5, 11 and 22 minutes after the first refusal, and nothing retries it when capacity frees. So it can start some minutes after capacity frees, not at once: about 5.5 minutes after the first refusal in our test. If capacity frees after about 22 minutes, the held deployment gets no further attempt before the 30-minute scheduling deadline and fails; deploy it again. On full hardware, stop one deployment before you start the next.
- Kamiwaza checks each VRAM share on its own, while the quota caps the bucket's total, so a share that fits alone can wait with the quota's reason.
- Running pods are never evicted, and a pod that requests no accelerator is never counted. Any quota your platform sets still applies.

Render and apply the quota again whenever you publish a profile, change GPU hardware, MIG layout, the device plugin or the VRAM plugin, or recreate the namespace. A resource that appears after the snapshot, such as a new MIG profile, is not limited until you do. If the signed file is no longer at hand, read the published profile back and take a fresh snapshot, then run the render and apply command above again:

```
kubectl -n kamiwaza get configmap "$PROFILE_CONFIGMAP" -o json >
"$PROFILE_CONFIGMAP.json"
kubectl get nodes -o json > nodes.json
```

The quota does not count a pod that selects GPUs through `NVIDIA_VISIBLE_DEVICES` without requesting them; GPU nodes must refuse such requests from unprivileged containers (see GPU Nodes (see online documentation)). It does not count Dynamic Resource Allocation claims either. When more than one node advertises the same VRAM bucket, one pod can take more than its `totalUnits` of one node's bucket, as long as the namespace's total stays under the quota; hold a per-node limit with admission policy.

To remove the quota, delete it as the cluster owner, and the `kamiwaza-extensions` quota too if you applied one:

```
kubectl -n kamiwaza delete resourcequota kamiwaza-owner-accelerators
kubectl -n kamiwaza-extensions delete resourcequota kamiwaza-owner-accelerators
```

Step 6: Set the Helm values

Add these values to the install's values file and upgrade the release with it:

```
core:
  inferenceResources:
    enabled: true
    bundleConfigMap: kamiwaza-inference-profiles-r1   # the name from step 5
    ownerPublicKey: <OWNER_PUBLIC_KEY>                 # publicKey from step 1
    clusterBindingId: <CLUSTER_BINDING_ID>             # from step 2
```

Helm value	Notes
`bundleConfigMap`	Kamiwaza reads the profile only from the ConfigMap with this name. Left at the default, `kamiwaza-inference-profiles`, it cannot read a renamed revision. On a namespace without an anchor (see online documentation), if an older ConfigMap with the default name still exists, Kamiwaza uses that older profile, with no error, for as long as it verifies.
`ownerPublicKey`	The canonical base64 of the raw 32-byte public key, as `keygen` and `pubkey` print it.
`clusterBindingId`	Must equal the profile's `clusterBindingId`.

On an anchored namespace, the anchor's values replace these three and `minimumBundleVersion`. Keep them equal to the anchor anyway: a Kamiwaza release from before the anchor ignores it, so rolling Kamiwaza back returns to these values.

On an install whose `global.imagePullSecrets` names the pull Secret, these four values are enough; an install that pulls every image without credentials needs no pull Secret. The staging values default as follows, and an explicit value always wins over its default:

Value	Default with tenant-mode inference on
`gpuRegistry`, `cpuRegistry`	The in-cluster model registry, which the model downloader already pushes to. It serves plain HTTP, so `gpuRegistryInsecure` and `cpuRegistryInsecure` default to `true` with it.
`gpuStagingImage`, `cpuStagingImage`	The chart's own model-fetch image, relocated like every chart image.
`gpuRegistryCredentialsSecret`, `cpuRegistryCredentialsSecret`	The first entry of `global.imagePullSecrets`, the install's pull Secret. Not defaulted when `global.openshift.serviceAccountImagePullSecretsOnly` is `true`.

Engine pods carry only this staging credential, as their image pull Secret: the engine and model-fetch images pull with it. Set `gpuRegistryCredentialsSecret` and `cpuRegistryCredentialsSecret` yourself only when your mirror needs different credentials from the install's pull Secret.

Deployments created before you turn tenant-mode inference on carry no tenant resources, so they cannot start again once it is on. After the upgrade, stop each one and deploy it again (see Deploy through the API (see online documentation)); the new deployment takes its resources from the signed profile and its image from the engine

catalog. While `core.context.embedding.autoProvision` is on (the default, except on `global.platform: osgm`), the platform's embedding model needs only the stop: the platform deploys it again on the next embedding request. With it off, deploy the embedding model again yourself.

While `core.context.embedding.autoProvision` is on, turning tenant-mode inference on for an install with no earlier embedding deployment also deploys the platform's bundled embedding model, all-MiniLM-L6-v2, on CPU, so expect one CPU engine pod shortly after the upgrade. On an install that already served the embedding model before you turned tenant-mode inference on, the platform finds that earlier deployment, still `DEPLOYED`, and deploys nothing until you stop it and an embedding request arrives. With it off, the platform does not deploy the embedding model on its own.

Step 7: Verify

Deploy a model that fits a GPU, from the Models page, then check the engine pods. The columns are the pod, its node, its RuntimeClass, the NVIDIA resources it was granted, and whether the grant was verified:

```
kubectl -n kamiwaza get pods -l kamiwaza.ai/deployment-id -o json | jq -r '.items[]
  | [.metadata.name, .spec.nodeName, (.spec.runtimeClassName // "-"),
     ([.spec.containers[].resources.limits // {} | to_entries[]
       | select(.key | startswith("nvidia.com/")) | "\(.key)=\(.value)"]
      | join(",") | if . == "" then "-" else . end),
     ([.status.conditions[]? | select(.type == "kamiwaza.ai/AllocationVerified") |
.status]
      | first // "-")]
  | @tsv'
```

```
tmi-3f2a9c...   gpu-node-1   nvidia   nvidia.com/gpu=1           True
tmi-8b1e04...   gpu-node-1   nvidia   nvidia.com/mig-3g.40gb=1   True
tmi-51c7aa...   gpu-node-1   -        -                          True
```

Expect the `nvidia` RuntimeClass and the GPU or MIG resource the deployment asked for on a GPU pod, and no RuntimeClass or NVIDIA resource on a CPU pod. Every pod should read `True` in the last column. Readiness alone is not proof of the requested grant: the `kamiwaza.ai/AllocationVerified` condition is.

A CPU pod requests the engine's CPU floor from the engine catalog, `100m` for llama.cpp and `500m` for whisper.cpp, and memory of at least 1.25 times the model's weights, rounded up to a whole GiB and never below the floor of `1Gi` for llama.cpp or `2Gi` for whisper.cpp. It sets requests only, no limits.

Recipe catalogs (optional)

A recipe catalog pins a specific model and configuration to exact settings. Without one, every deployment is derived. Render each catalog with the owner tool, which validates it with the parser Kamiwaza runs when it mounts it and emits an immutable ConfigMap whose name carries the catalog's digest:

```
owner_tool render-catalog --catalog gpu-recipes.json \
  --namespace kamiwaza --capability gpu > gpu-catalog.json
kubectl apply --dry-run=server -f gpu-catalog.json
kubectl apply -f gpu-catalog.json
export GPU_CATALOG_CONFIGMAP="$(jq -r '.metadata.name' gpu-catalog.json)"
```

Use `--capability cpu` for a CPU catalog. Point the release at the exact name with `core.inferenceResources.gpuRecipeCatalogConfigMap` or `cpuRecipeCatalogConfigMap`. Publish a new revision for every change and never mutate a published ConfigMap. Keep the previous revision until every consumer has rolled out; roll back by selecting the previous name.

A pinned deployment runs the image its recipe variant names, and the profile binding's `runtime.image` must be the same reference. Kamiwaza does not relocate it: the bundle's image map, `global.imageRegistryPrefix`, and `global.airgap.registry.url` apply only to the engine catalog's images. In a mirrored or air-gapped install, name your mirror's copy, by digest, in both the recipe variant and the binding.

Engine images

The engine catalog in the core image pins every engine image by digest. Kamiwaza relocates those references once, when it loads the catalog, using the same values the chart relocates its own images with:

1. `core.inferenceResources.engineImageMap`, when set. The tenant OCI bundle writes it for you; see Bundle installs (see online documentation).
2. Otherwise `global.imageRegistryPrefix`: `<prefix>/images/<name>@<digest>`, where `<name>` is the last segment of the catalog's repository.
3. Otherwise `global.airgap.registry.url`: `<registry>/<catalog repository>@<digest>`.
4. Otherwise the reference as the catalog pins it.

An engine image must therefore be in your registry **by digest**, at the location the rule produces. Most catalog pins are untagged manifests, so a copy by tag does not bring them. Print the pins with the catalog command in step 4 (see online documentation), and see Install Without Internet Access (see online documentation) for the copy.

Engine pods pull with the staging registry Secret, `core.inferenceResources.gpuRegistryCredentialsSecret` and `cpuRegistryCredentialsSecret`, which default to the install's pull Secret; see step 6 (see online documentation). There is no separate engine pull-secret setting. Set those two values only when your mirror needs different credentials.

A deployment that a recipe pins is the exception to the relocation above: it runs the image the owner signed, exactly as written. See Recipe catalogs (see online documentation).

After you change any of these values, redeploy the affected inference deployments: stop each one and deploy it again (see Deploy through the API (see online documentation)). A running deployment keeps the engine reference it was planned with, and so does the platform's embedding model. Stop the embedding model's deployment too. While `core.context.embedding.autoProvision` is on, leave the redeploy to the platform: it deploys the model again on the next embedding request, such as `POST /api/embedding/generate`, and not before. That request returns HTTP 503 with the code `embedding_deploying` and a `retry_after_seconds` value, and embedding requests return it until the new deployment is ready, about 75 seconds on an air-gapped A100 install; retry them. With it off, the platform does not redeploy the model, and embedding requests return HTTP 503 with the code `embedding_unavailable` until you deploy it again yourself. On an upgrade, keep the previous release's engine images in your registry until every deployment has been redeployed; removing them breaks rescheduling.

Bundle installs

A tenant OCI bundle carries the engine images for one architecture, and writes `values/99-engine-images.yaml` with the map that tells Kamiwaza where it imported each one. Keep that file last among the install's values files. A cluster with amd64 and arm64 nodes needs a bundle for each architecture.

A bundle can leave engines out. An engine the bundle does not carry is refused at admission, before the deployment record exists:

```
cannot derive inference resources for this deployment: engine 'vllm' is not qualified
in this release
```

Attest the NVIDIA driver's CUDA major

For each NVIDIA binding, the platform selects the newest image of the engine that both the card and the host driver can run. The card's limit is the `computeCapability` the binding attests. The driver's limit is `cudaDriverMajor`, an optional integer from 11 to 99 in the same `accelerator` block: the major of the `CUDA Version` that `nvidia-smi` prints in its header on the host (e.g., `12` on an R570 driver, which reports CUDA 12.8, and `13` on an R580 driver, which reports CUDA 13.0). An image is selected only when its CUDA major is at or below the attested value.

The detector in step 3 (see online documentation) attests `cudaDriverMajor` by default. Without it, the platform does not judge the driver: a Turing or newer card on an R570 host is given the CUDA 13 llama.cpp image, which exits at startup with `CUDA driver version is insufficient for CUDA runtime version`. Re-detect and publish a new signed profile revision after any driver change, including a driver rollback.

A binding without `cudaDriverMajor` keeps verifying and selects the newest image its card can run:

- llama.cpp resolves its CUDA 13 image on compute capability 7.5 or later, and its CUDA 12 image on Volta (7.0).
- vLLM resolves its CUDA 13 image on amd64 on compute capability 8.0 or later, because the engine also requires `bf16`, and on arm64 only on a GB10 (12.1).
- Diffusion resolves its one image, CUDA 12, on compute capability 8.0 or later, on amd64 only.
- A Pascal (6.x) or older card is refused.

vLLM ships only a CUDA 13 image. On a binding that attests `cudaDriverMajor: 12` the platform refuses it:

```
host driver supports CUDA 12 (attested cudaDriverMajor); engine vllm ships nvidia/amd64
images for CUDA 13+
```

On a binding that does not attest the fact, the platform cannot tell, and the vLLM image fails at startup on a CUDA 12 driver. When the card is older than every image, the reason names the compute capability instead:

```
llamacpp requires compute capability 7.0; this binding declares 6.1
```

A deployment without an `inferenceResources` block is refused with HTTP 422 before the deployment record exists, with the reason after `cannot derive inference resources for this deployment:`. This happens only when the driver or the card rules out every image on every GPU binding in the profile. A request that carries its own block is refused at binding selection: the deployment fails with `NoFit` and the reason in its error message. To resolve a driver refusal, upgrade the host driver, re-detect, and publish a new signed profile revision. No owner action resolves a card refusal, because the release ships no image for that card.

Before you roll Kamiwaza back to a release earlier than 1.3.0, publish a new signed profile revision without `cudaDriverMajor` and point the release at it. Earlier releases refuse the unknown key and reject the whole profile, so every tenant deployment, CPU deployments included, fails authorization.

Requests that name their own resources

A deployment from the Models page needs no `inferenceResources` block: the platform derives one. To choose the profile, device count or memory yourself, deploy through the API, `POST /api/serving/deploy_model`, with the block in the request body. The block names the logical profile and capability rather than a Node or a provider-specific inventory:

```
{
  "inferenceResources": {
    "schemaVersion": 1,
    "accelerator": {
      "capability": "gpu",
      "count": 1,
      "memory": {"minimum": "1Gi"},
      "isolation": "any-qualified",
      "profile": "nvidia-whole"
    },
    "runtime": {"selection": "automatic"}
  }
}
```

For `cpu-standard`, use the CPU request shape instead:

```
{
  "inferenceResources": {
    "schemaVersion": 1,
    "cpu": {
      "architecture": "amd64",
      "requests": {"cpu": "2", "memory": "4Gi"}
    },
    "runtime": {"selection": "automatic"}
  }
}
```

Do not add Node names, Node selectors owned by the tenant, or cluster-scoped resources to the tenant request.

Deploy through the API

Get a token as an administrator. Set `KAMIWAZA_URL` to your Kamiwaza domain; if the edge certificate is issued by a private authority, add `--cacert <CA_FILE>` to each `curl` command. The `admin` password is the one from your install page's verify step:

```
KAMIWAZA_URL="https://<KAMIWAZA_DOMAIN>"
printf 'admin password: '; read -rs ADMIN_PASSWORD; echo   # typed, not echoed
# curl reads the password from stdin, so it never appears on a command line.
TOKEN=$(printf '%s' "$ADMIN_PASSWORD" | curl -sS -X POST "$KAMIWAZA_URL/api/auth/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  --data-urlencode "username=admin" --data-urlencode "password@-" \
  | jq -r '.access_token')
unset ADMIN_PASSWORD
```

Find the model, its configuration and its weights file. A model is in the list once it has been downloaded; see Downloading Models (see online documentation):

```
curl -sS -H "Authorization: Bearer $TOKEN" "$KAMIWAZA_URL/api/models/" \
  | jq -r '.[] | [.id, .name] | @tsv'
MODEL_ID="<MODEL_ID>"
curl -sS -H "Authorization: Bearer $TOKEN" "$KAMIWAZA_URL/api/model_configs/?
model_id=$MODEL_ID" \
  | jq -r '.[] | [.id, .name, .default] | @tsv'
curl -sS -H "Authorization: Bearer $TOKEN" "$KAMIWAZA_URL/api/models/$MODEL_ID" \
  | jq -r '.m_files[] | [.id, .name] | @tsv'
```

Write the request, with the configuration marked `true` in the second list and the weights file to serve, then deploy:

```
cat > deploy.json <<'JSON'
{
  "m_id": "<MODEL_ID>",
  "m_config_id": "<CONFIG_ID>",
  "m_file_id": "<FILE_ID>",
  "inferenceResources": {
    "schemaVersion": 1,
    "accelerator": {
      "capability": "gpu",
      "count": 1,
      "memory": {"minimum": "8Gi"},
      "isolation": "any-qualified",
      "profile": "nvidia-whole"
    },
    "runtime": {"selection": "automatic"}
  }
}
JSON
curl -sS -X POST -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json"
\
  --data @deploy.json "$KAMIWAZA_URL/api/serving/deploy_model"
```

```
"631e7142-3417-487c-be65-d7715ba6cbf2"
```

The response is the deployment's ID, returned before the launch finishes. Follow its status until `DEPLOYED`; see Model Deployment (see online documentation) for the statuses:

```
curl -sS -H "Authorization: Bearer $TOKEN" \
  "$KAMIWAZA_URL/api/serving/deployment/<DEPLOYMENT_ID>/status"; echo
```

To stop a deployment, e.g. before deploying it again after a registry change:

```
curl -sS -X DELETE -H "Authorization: Bearer $TOKEN" \
  "$KAMIWAZA_URL/api/serving/deployment/<DEPLOYMENT_ID>"; echo
```

The response is `true`, and the deployment's status becomes `STOPPED`.

Upgrade and recover

To change the profile, publish a new revision under a new name with a higher `version`. If the namespace has an accelerator quota (see online documentation), render and apply it from the new revision with `owner_tool quota` before you anchor it, so the old quota never refuses a deployment the new profile allows. Then anchor it (see online documentation). The anchor takes effect at the next deployment, with no upgrade or restart. Point

`bundleConfigMap` at the new revision too, upgrade the release, and repeat the checks in step 7 (see online documentation). On a namespace without an anchor, pointing `bundleConfigMap` at the new revision and upgrading is what changes the profile.

To roll back on an anchored namespace, the cluster owner re-signs the previous profile at a higher `version` and anchors that, or anchors the previous revision itself, which lowers the minimum version to its `version`. Pointing `bundleConfigMap` at an older revision has no effect there. On a namespace without an anchor, point `bundleConfigMap` at the previous revision.

If a create, readiness, authorization, or deadline step fails, the platform marks the allocation failed and removes only resources owned by that attempt; keep the prior known-good workload until the replacement passes its checks.

Upgrading Kamiwaza, including its core image, does not require a new signed profile. Sign a new revision only when your profiles change, or when the release notes say the profile schema changed.

To rotate the owner key, create a new key, sign a new revision with it, and anchor that revision with `owner_tool anchor --public-key <NEW_PUBLIC_KEY>`; the annotation changes the key, the profile name, and the minimum version together. Then set `ownerPublicKey` and `bundleConfigMap` together in one upgrade. On a namespace without an anchor, the upgrade is what rotates the key.

The anchor stops a superseded or foreign profile from being accepted. It does not stop a namespace administrator who can change Kamiwaza's own workloads, or turn tenant-mode inference off in the Helm values, from running GPU workloads without a profile. Hold that boundary as the cluster owner, with the accelerator quota (see online documentation) and with admission policy. After any change to GPU hardware, MIG layout or device plugins, render and apply the quota again with `owner_tool quota`. A quota limits GPUs only on nodes that refuse GPU requests by environment; see GPU Nodes (see online documentation).

Turning that setting on for a node that already serves GPUs changes how pods get their GPU. Follow the order in Turn the setting on for a node that serves GPUs (see online documentation), which includes redeploying every GPU deployment before the node refuses the environment: a deployment whose pod holds its GPU by environment only starts without one when its container next restarts after the change.

Troubleshooting

A GPU deployment returns HTTP 422 with `Core cannot see a GPU on this install`. Tenant-mode inference is off. Turn it on as described in Turn on tenant-mode inference (see online documentation).

A deployment returns HTTP 503 with `owner profile bundle: invalid inference profile authorization:` and a reason. Kamiwaza found the profile but refused it. The reason names the check:

Reason	Fix
`bundle did not arrive from the configured ConfigMap`	`bundleConfigMap` does not name the published revision, or it is in another namespace.
`bundle targets another cluster binding or namespace`	Compare the profile's `clusterBindingId` and `namespace` with the anchor, or on a namespace without one, with the Helm values.
`bundle version is below the minimum bundle version`	The profile is older than the anchored minimum, or `minimumBundleVersion` on a namespace without an anchor. Publish and anchor a newer revision.
`bundle is signed by a key this cluster does not trust`	The anchor's keys, or `ownerPublicKey` without an anchor. do not include the key that signed the profile. Print it with `owner_tool pubkey`.
`the namespace's owner anchor is malformed`	The `inference.kamiwaza.ai/owner-anchor` annotation is not what `owner_tool anchor` writes. The cluster owner re-applies it.
`the configured namespace is not this pod's namespace`	`KAMIWAZA_K8S_NAMESPACE` names another namespace. Reinstall with the release's own values.
`bundle is expired or its window is empty`	Publish a new revision with a later `expiresAt`.

A deployment returns HTTP 422 with `engine '<name>' is not qualified in this release`. The engine is not in the engine catalog on this install: MLX is never in it, and a bundle install may have left the engine out.

A deployment returns HTTP 422 naming the driver's CUDA major or the card's compute capability. See Attest the NVIDIA driver's CUDA major (see online documentation).

A deployment returns HTTP 503 with `owner profile bundle: the namespace's owner anchor could not be read`. Kamiwaza cannot read its own namespace, so it refuses rather than trusting the Helm values. Upgrade the release so the chart restores the read, and check that the installer still holds the `admin` role in the namespace.

A deployment stays pending with `exceeded quota: kamiwaza-owner-accelerators` and a `requests.<resource>`. The namespace's accelerator quota (see online documentation) is full for that resource, usually because another deployment holds it. Stop that deployment. This one starts at Kubernetes' next retry after that pod is gone, which can be up to about 11 minutes later. If it has been held for more than about 22 minutes, it gets no further retry before the scheduling deadline and fails; deploy it again. If the quota for the resource reads `0`, either a `--limit` set it, or the profile does not grant it: render the quota without that limit, or publish a profile that grants the resource, then apply the quota again.

A request with its own block fails with `NoFit`. No binding in the profile holds the request: compare its memory, count, and isolation with the profile's bindings.

A deployment stays `REQUESTED` with no error, and its engine pod is `Pending`. The pod cannot be scheduled, and the deployment reports no error until the per-attempt scheduling deadline, `schedulingTimeoutSeconds`, passes. A tensor-parallel deployment that needs two whole GPUs while another deployment holds one of them is a common cause. Find the pod and read its events:

```
kubectl -n kamiwaza get pods -l kamiwaza.ai/deployment-id --field-
selector=status.phase=Pending
kubectl -n kamiwaza describe pod <POD_NAME> | sed -n '/Events:/,$p'
```

`Insufficient nvidia.com/gpu` means no node has enough free GPUs of the requested kind; stop a deployment that holds one, or add GPUs. `had untolerated taint` means a GPU node is tainted; see GPU Nodes (see online documentation).

An engine pod fails with `ImagePullBackOff`. Read the image it pulls:

```
kubectl -n kamiwaza get pod <POD_NAME> -o jsonpath='{.spec.containers[*].image}'; echo
```

An image that names a registry you do not mirror means the relocation values are not set; see Engine images (see online documentation). An image in your registry means that digest was not copied there, or the staging registry Secret does not hold its credentials.

A deployment returns HTTP 503 with `engine image map carries no image from this release's catalog`. The bundle's map was built for another release. Build the bundle for the core image you install.

Support status

Support covers the configurations marked Supported in Compatibility scope (see online documentation), qualified on k0s Kubernetes 1.36, within the limits each row states. Anything the table does not mark Supported, such as AMD GPUs or a deployment that spans more than one device, is not supported in this release.

Model Placement Overview

> **NOT IN TENANT MODE**
>
> This page describes placement on installs where Kamiwaza can read the cluster's Nodes. A namespaced install serves local models through tenant-mode inference (see online documentation) instead. Placement there follows the bindings in the cluster owner's signed profile, not the node labels, VRAM budgets, and notices this page describes.

When you deploy a model, Kamiwaza decides where it runs: which node, which GPU (or memory pool), and how much memory to reserve for it. This is **model placement**. Placement is automatic — you deploy a model the same way on a datacenter cluster, a single-GPU workstation, or a MacBook, and Kamiwaza picks an appropriate home for it.

The goal of placement is safe GPU coexistence: running more than one model on the same hardware without them fighting over memory.

Why GPU coexistence matters

A GPU is usually much larger than any single model you put on it. A 6 GB model deployed exclusively on an 80 GB card leaves 74 GB idle. Without placement, every deployment claims a whole GPU, so a second small model on the same card is refused even though there is plenty of room.

With placement, Kamiwaza estimates each model's memory footprint, reserves that amount on a specific GPU, and lets additional models use the remaining capacity. A model that does not fit anywhere fails fast with a structured placement error recorded on the deployment — it is never left silently pending. See Fractional GPU Serving (see online documentation) for how the budgeting works.

Where Kamiwaza runs

Placement adapts to the way Kamiwaza is installed. You will see the active topology on each deployment's details page.

Topology	What it looks like	Examples
Managed cluster	Kamiwaza is installed into a namespace on your existing Kubernetes cluster. Your cluster admin has already installed the NVIDIA or AMD GPU Operator, and placement respects whatever GPU sharing the admin configured (MIG partitions, time-slicing, MPS).	OpenShift, HPE Private Cloud AI, customer-installed Kubernetes
Standalone cluster	Kamiwaza owns the cluster install on hardware you point it at, including the GPU device exposure.	NVIDIA DGX Spark workstations, AMD Strix Halo workstations, edge appliances
macOS	A standalone install on Apple Silicon. The Kamiwaza control plane runs in a lightweight VM; the model processes themselves run natively on the host so they get full Metal GPU access.	MacBook and Mac Studio (M-series)

On managed clusters, Kamiwaza never edits node labels or requires privileged access — the one-time cluster-admin steps at install are limited to a custom resource definition and a read-only role for node labels.

Three kinds of GPU hardware

Placement treats GPUs differently depending on how their memory is built and shared. Three practical examples cover the range:

A datacenter card with hardware partitioning. An NVIDIA H100 with MIG (Multi-Instance GPU) enabled is carved into hardware slices, each with its own dedicated memory. Kamiwaza places one model per slice. A crash or memory spike in one slice cannot touch a model in another — the isolation is enforced by the hardware itself.

A discrete card without partitioning. An NVIDIA T4, L4, or an A100 with MIG disabled has one fixed pool of VRAM on the card. There is no hardware fence, so Kamiwaza shares the card by accounting: each model reserves a memory budget in GB, and models coexist as long as their combined budgets fit the card. Two 6 GB models fit comfortably on a 16 GB T4.

A machine with unified memory. A MacBook, an AMD Strix Halo workstation, or an NVIDIA DGX Spark has no separate VRAM at all — the CPU and GPU share one system memory pool. Kamiwaza budgets model deployments against that shared pool (leaving headroom for the operating system) and lets concurrent models share the GPU through the OS scheduler.

Kamiwaza names these three situations **hardware classes**:

Hardware class	Example hardware	How models share
Hardware-isolated	H100/A100 with MIG; AMD Instinct MI300 with partitioning	One model per hardware partition
Software-shared	T4, L4, A100/H100 without partitioning	Per-GPU memory budgets (fractional), or whole-GPU exclusive
Unified memory	Apple Silicon, AMD Strix Halo, NVIDIA DGX Spark (GB10)	Budgets against the shared system memory pool

The class is detected automatically from node labels — you never select it. GPU Hardware Classes (see online documentation) explains each class, its isolation guarantees, and the signals Kamiwaza reads.

What you see after deployment

Every deployment's details surface where it landed and what it reserved:

- **Topology** — managed cluster or standalone cluster (macOS installs appear as standalone with the `metal_spawner` sharing class).
- **Node and GPU** — the node name and GPU index (where applicable).
- **Hardware class and sharing class** — for example `unified_memory`, or `mig_2g_20gb` for a model placed in a MIG slice.
- **Allocated capacity (GB)** — the memory budget reserved for this deployment.

See the Placement Deployment Guide (see online documentation) for reading these fields and troubleshooting placement.

See also

- GPU Hardware Classes (see online documentation) — the class taxonomy in depth
- Fractional GPU Serving (see online documentation) — memory budgeting and multi-model-per-GPU
- Placement Deployment Guide (see online documentation) — step-by-step deployment and troubleshooting
- Model Deployment (see online documentation) — the general deployment lifecycle and statuses

GPU Hardware Classes

> **NOT IN TENANT MODE**
>
> This page describes placement on installs where Kamiwaza can read the cluster's Nodes. A namespaced install serves local models through tenant-mode inference (see online documentation) instead. Placement there follows the bindings in the cluster owner's signed profile, not the node labels, VRAM budgets, and notices this page describes.

Kamiwaza model placement specializes its behavior by **hardware class** — a classification of how a GPU's memory is built and how it can be shared. The class determines which placement primitive Kamiwaza uses, how many models can share a device, and what isolation you can expect between them.

If you have not read it yet, start with the Model Placement Overview (see online documentation), which introduces the three classes through practical examples.

The class table

Class	Members	Placement primitive	Isolation
Hardware-isolated	NVIDIA MIG-capable cards (A100, H100, B100) with MIG enabled; AMD Instinct MI300+ with partitioning enabled	One model per hardware partition (for example, a MIG slice requested as `nvidia.com/mig-2g.20gb`)	Hardware: dedicated memory and fault isolation per partition
Software-shared	Discrete-memory GPUs without hardware partitioning: T4, L4, A100/H100 with MIG disabled, and similar; also covers admin-configured time-slicing or MPS on managed clusters	Fractional per-GPU memory budgets in GB, or the cluster's configured sharing strategy, or whole-GPU exclusive	Software: memory budgets are enforced when the model is scheduled; co-located models share compute and faults
Unified memory	Apple Silicon (M-series), AMD Strix Halo, NVIDIA DGX Spark (GB10 Grace-Blackwell)	Budget against the aggregate system memory pool	Process-level only: concurrent models share the GPU through the OS scheduler

In deployment details these appear as the `hardware_class` values `hardware_isolated`, `software_shared`, and `unified_memory`.

Hardware-isolated

When a cluster admin enables MIG on an NVIDIA card (or partitioning on an AMD Instinct accelerator), the card is divided into fixed partitions, each with its own memory and fault domain. Kamiwaza picks a partition profile that fits the model's estimated footprint and requests it — for example, a 2g.20gb MIG slice for a model estimated under 20 GB.

What to expect:

- **One model per partition.** Each deployment gets a whole slice to itself. Kamiwaza does not pack multiple models into one slice.
- **True isolation.** A crash, memory leak, or GPU fault in one partition does not affect models in other partitions.
- **Capacity is the partition size**, not the whole card. A model that needs more than the largest configured partition will not fit — the deployment fails with a structured no-fit error (see What a NoFit error means (see online documentation)).
- The sharing class on deployment details names the partition, e.g., `mig_2g_20gb`.

Software-shared

A discrete GPU without hardware partitioning has one VRAM pool and no hardware fences. Kamiwaza shares it by accounting:

- **Fractional memory budgets.** Each deployment reserves an estimated footprint in GB on a specific GPU. Models coexist on a card as long as their combined budgets fit. This is the default on standalone clusters and is described in detail in Fractional GPU Serving (see online documentation).
- **Cluster-configured sharing.** On a managed cluster where the admin has configured time-slicing or MPS through the GPU Operator, Kamiwaza requests the shared resource the cluster advertises and the configured strategy governs sharing.
- **Whole-GPU exclusive.** When neither of the above applies (or fractional placement is disabled), a deployment claims the entire GPU and no other model can join it.

What to expect:

- Memory budgets are checked **when the model is scheduled** — a model that would overflow the card is refused up front.
- The serving engines self-limit to their allocation (for example, vLLM's GPU memory utilization setting), but there is **no compute or fault isolation** between models sharing a card: a misbehaving workload can slow or destabilize its neighbors.
- On a managed cluster with a GPU Operator installed but **no sharing strategy configured**, Kamiwaza places whole-GPU and attaches a `SharingNotConfigured` notice to the deployment so you know density is limited to one model per GPU until the admin enables sharing.

Unified memory

Apple Silicon, AMD Strix Halo, and NVIDIA DGX Spark machines have a single memory pool shared by the CPU and GPU. There is no separate VRAM to slice, so unified-memory placement is its own algorithm, not a one-slot variant of discrete-GPU placement:

- Deployments are budgeted against the **aggregate system memory**, with headroom held back for the operating system and other processes.
- Concurrent models share the GPU through the OS scheduler — there is no partitioning of compute.
- On macOS, the model processes run **natively on the host** (not inside the cluster) so they get full Metal GPU access; the sharing class for these deployments is `metal_spawner`.

- Memory exhaustion on these machines includes everything else the host is doing. If the system is busy, a model that fit on paper can still run out of memory at load time — see Limits to know (see online documentation).

How Kamiwaza detects the class

Detection is automatic, driven by node labels — you never set the class yourself.

- **Managed clusters:** Kamiwaza reads the labels your GPU Operator already publishes, such as `nvidia.com/gpu.product` and `nvidia.com/mig.capable` (NVIDIA) or `amd.com/gpu.cu-count` and `dcm.amd.com/gpu-config-profile` (AMD). Kamiwaza never adds or edits node labels on a managed cluster.
- **Standalone clusters:** the Kamiwaza installer publishes its own labels during hardware detection, including:

Label	Meaning
`kamiwaza.ai/gpu-memory-class`	`unified` or `discrete` — the unified-memory signal
`kamiwaza.ai/gpu-memory-mb`	Detected GPU (or shared-pool) memory. The value is in MiB, despite the label name
`kamiwaza.ai/gpu-vendor`	GPU vendor

On a standalone cluster you can inspect the labels directly:

```
kubectl get node <node-name> -o jsonpath='{.metadata.labels}' | tr ',' '\n' | grep gpu
```

If a node's GPU labels are incomplete, the deployment surfaces a `PartialDiscovery` notice naming the missing labels, and placement proceeds with conservative assumptions (for example, sharing is treated as not configured). A node whose capacity labels are missing cannot be budgeted, so it will not win placement until those labels are present.

See also

- Model Placement Overview (see online documentation) — the classes introduced by example
- Fractional GPU Serving (see online documentation) — how software-shared and unified-memory budgeting works
- Placement Deployment Guide (see online documentation) — deploying and troubleshooting

Fractional GPU Serving

> **NOT IN TENANT MODE**
>
> This page describes placement on installs where Kamiwaza can read the cluster's Nodes. A namespaced install serves local models through tenant-mode inference (see online documentation) instead. Placement there follows the bindings in the cluster owner's signed profile, not the node labels, VRAM budgets, and notices this page describes.

Fractional GPU serving lets multiple models share one GPU, each with its own reserved memory budget. Instead of claiming a whole card, a deployment reserves the number of gigabytes it is estimated to need; the remaining capacity stays available for other models. This page explains when fractional serving applies, how the budgeting works, and how to read the error you get when a model does not fit.

When fractional serving applies

Fractional serving is governed by the hardware class (see online documentation) of the GPU:

Hardware	Fractional?	How sharing works
Discrete GPU without partitioning (T4, L4, A100/H100 with MIG disabled)	Yes	Per-GPU memory budgets in GB
Unified memory (Apple Silicon, Strix Halo, DGX Spark)	Yes	Budget against the shared system memory pool
MIG / hardware partitions	No	One model per partition; the partition itself is the unit of sharing
Any GPU with fractional placement disabled	No	Whole-GPU exclusive — one model per card

Fractional placement is enabled by default. Installations with stricter compliance requirements can disable it at install time with the Helm value `placement.vramPluginV2.enabled=false` on the placement-operator chart, which falls back to whole-GPU exclusive placement: the first model claims the card, and a second model on the same card fails with reason `insufficient_capacity`.

How the VRAM budget works

When you deploy a model, Kamiwaza estimates its memory footprint — weights, context/KV cache, and per-deployment overhead. (The SDK exposes this same estimator as `client.serving.estimate_model_vram()` if you want to preview a deployment request's footprint.)

The estimate becomes a reservation against a specific GPU. Each GPU on a node exposes its capacity as a per-device resource named `kamiwaza.ai/vram-gb-gpu-<i>` (where `<i>` is the GPU index), measured in GB. Kubernetes enforces these budgets when the model is scheduled, so an over-budget model is refused **before** anything starts — it is never left half-running or silently pending.

Details worth knowing:

- **One budget per physical GPU.** A node with four cards exposes four independent budgets (`...-gpu-0` through `...-gpu-3`); each model lands on exactly one card's budget.
- **Unified-memory machines expose a single budget** (`kamiwaza.ai/vram-gb-gpu-0`) sized to the shared memory pool, since the CPU and GPU draw from the same pool.
- **Unified-memory machines hold back an 8 GiB operating-system reserve** from the shared pool before the budget is advertised, so deployments cannot starve the host. Discrete-GPU budgets are sized to the card's detected VRAM.

On a standalone cluster you can see the advertised budgets directly:

```
kubectl get node <node-name> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
vram-gb-gpu
```

Example: packing models on one GPU

A 16 GB NVIDIA T4 with two 6 GB models:

1. Deploy model A (estimated 6 GB). It reserves 6 GB on `gpu-0`. Remaining budget ≈ 10 GB.
2. Deploy model B (estimated 6 GB). It fits the remaining budget and lands on the same card. Both deployments reach `DEPLOYED` and serve traffic concurrently.
3. Deploy model C (estimated 6 GB). The remaining budget (≈ 4 GB) is too small. The deployment fails fast with the structured no-fit reason `insufficient_capacity` — no pending pod, no partial deployment.

On a node with mixed cards (say, an 8 GB and a 24 GB GPU), placement picks a card whose remaining budget fits the request, so a 10 GB model goes to the 24 GB card even if the 8 GB card is idle.

Models larger than one GPU

A model whose footprint exceeds any single GPU can be deployed with tensor parallelism across multiple GPUs **on one node**: the request reserves a per-shard budget on each of N cards. If no node has N GPUs with enough free capacity, the request is rejected with `insufficient_gpu_count`. Multi-node parallelism is not supported in this release.

What a NoFit error means

The deploy API is asynchronous: it accepts the request and returns the deployment ID immediately. When no eligible GPU (or set of GPUs) can satisfy the request, the deployment then fails fast — before any pod is created — instead of sitting pending. The deployment shows status `FAILED` with `last_error_code` set to

`NoFitError` and the no-fit reason recorded in `last_error_message`. (When deploying through the SDK with the default `wait=True`, this surfaces as a `DeploymentFailedError`; see the Placement Deployment Guide (see online documentation).)

The reason tells you why the model did not fit:

Reason	Meaning	Common causes and what to do
`insufficient_capacity`	No single GPU (or partition) has enough free budget for the request	The model is too large for the hardware, or other deployments hold the budget. Choose a smaller or more quantized variant, reduce context length, or stop an unused deployment.
`insufficient_system_memory`	Unified-memory budget exhausted	The shared memory pool is fully reserved. Stop another deployment or pick a smaller model.
`insufficient_gpu_count`	A tensor-parallel request needs N GPUs but no node has N free	Lower the tensor-parallel size or free GPUs on a multi-GPU node.
`vendor_mismatch`	The selected engine requires a GPU vendor the cluster does not have	For example, an engine that requires CUDA on an AMD-only cluster. Pick a model/engine variant that matches your hardware.
`accel_version_unmet`	No node satisfies the engine's minimum accelerator version (for example, a required CUDA version)	Upgrade GPU drivers, or choose an engine/model variant built for your driver generation.
`sharing_not_configured`	A managed cluster has no GPU sharing configured and the model cannot fit as whole-GPU	Ask your cluster admin to enable a sharing strategy, or free a GPU.

Limits to know

Fractional budgets are **reservations, not real-time meters**:

- Budgets track what Kamiwaza deployments have reserved. Memory used by non-Kamiwaza processes on the same GPU or host is not tracked, so a model that fit at scheduling time can still hit an out-of-memory error at load time if something else consumed the memory — surfaced on the deployment as error code `OOM` (see troubleshooting (see online documentation)).
- The serving engines self-limit to their allocation (for example, vLLM's GPU memory utilization setting), but enforcement is cooperative on shared cards.
- Models sharing a GPU share compute and fault domains: there is no performance or crash isolation between them. If you need hard isolation, use hardware partitioning (hardware-isolated class (see online documentation)).

See also

- Model Placement Overview (see online documentation) — why coexistence matters
- GPU Hardware Classes (see online documentation) — which hardware gets which primitive
- Placement Deployment Guide (see online documentation) — deploying multi-model setups step by step
- Model Deployment Troubleshooting (see online documentation) — general deployment issues

Placement Deployment Guide

> **NOT IN TENANT MODE**
>
> This page describes placement on installs where Kamiwaza can read the cluster's Nodes. A namespaced install serves local models through tenant-mode inference (see online documentation) instead. Placement there follows the bindings in the cluster owner's signed profile, not the node labels, VRAM budgets, and notices this page describes.

Placement is automatic — deploying a model with placement is the same deploy flow you already use. This guide shows how to deploy multiple models onto shared GPUs, read where each deployment landed, and troubleshoot placement problems.

For background on how placement decides, read the Model Placement Overview (see online documentation) and Fractional GPU Serving (see online documentation).

Before you start

- A model downloaded and ready to deploy. See Downloading Models (see online documentation).
- A rough sense of the model's memory footprint versus your hardware. The UI shows VRAM guidance per variant; the SDK exposes `estimate_model_vram`.
- On a managed cluster: the NVIDIA or AMD GPU Operator installed by your cluster admin. If no sharing strategy is configured, deployments still work but density is limited to one model per GPU (see SharingNotConfigured (see online documentation) below).

Deploy a model in the UI

1. Navigate to the **Models** page and select the model.
2. Click **Deploy**. In Novice Mode, Kamiwaza picks a platform-appropriate variant with sensible defaults; in Advanced Mode you can select the engine and parameters yourself. See the GUI Walkthrough (see online documentation).
3. Kamiwaza estimates the model's footprint, picks a GPU (or memory pool) with enough free budget, and starts the deployment. No placement input is required.
4. Watch the status move through `DEPLOYING` and `INITIALIZING` to `DEPLOYED`. The statuses are described in Model Deployment (see online documentation).

To run a second model on the same hardware, just deploy it the same way. If the combined budgets fit, both models run side by side; if not, the second deployment fails fast with a NoFit error (see online documentation).

Deploy a model with the SDK

```python
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient(base_url="https://<your-host>/api")

# Deploy by model ID (a model already in your catalog). If m_config_id is
# omitted, the SDK resolves the model's default configuration for you.
# Placement is automatic.
deployment_id = client.serving.deploy_model(model_id=model_id)

# Or deploy by Hugging Face repo ID (the SDK looks up the catalog entry),
# pinning an explicit configuration with m_config_id.
deployment_id = client.serving.deploy_model(
    repo_id="Qwen/Qwen3-8B",
    m_config_id=config_id,
)

status = client.serving.get_deployment_status(deployment_id)
```

The deploy API is asynchronous: the server accepts the request and returns the deployment ID immediately. By default the SDK blocks client-side (`wait=True`), polling until the deployment reaches `DEPLOYED` ; it raises `DeploymentFailedError` if the deployment reaches a terminal failure state (`FAILED` , `ERROR` , or `MUST_REDOWNLOAD`) and `TimeoutError` if the deployment is not ready within `timeout_seconds` . Pass `wait=False` to get the deployment ID back as soon as the server accepts the request, then observe progress with `get_deployment_status` or `wait_deployment_ready` .

A model that cannot be placed fails fast with the no-fit reason recorded on the deployment (`last_error_code` , `last_error_message`) — see What a NoFit error means (see online documentation).

Monitor placement

Open a deployment's details to see where it landed:

Field	Meaning
`topology`	`managed_cluster` or `standalone_cluster` (macOS installs show the standalone topology with the `metal_spawner` sharing class)
`node_name`	The node the model was placed on
`gpu_index`, `gpu_vendor`	Which GPU on the node, and its vendor
`hardware_class`	`hardware_isolated`, `software_shared`, or `unified_memory` — see GPU Hardware Classes (see online documentation)
`sharing_class`	How the device is shared, e.g., `mig_2g_20gb`, `whole_gpu`, `unified_memory`, `metal_spawner`
`allocated_capacity_gb`	The memory budget reserved for this deployment, in GB

Verify

1. Confirm each deployment shows `DEPLOYED` in the UI.
2. Send a short test prompt to each deployment's endpoint (shown in the UI).
3. If you deployed multiple models to one GPU, confirm both respond — they are serving concurrently from the same card.

Troubleshooting

The deployment fails immediately with a placement (NoFit) error

This is placement telling you the model does not fit anywhere, before any pod starts. The deployment shows `FAILED` with `last_error_code` set to `NoFitError` and the no-fit reason in `last_error_message` (the SDK raises `DeploymentFailedError` when waiting). The NoFit reference table (see online documentation) lists every reason with common causes and fixes. The quick version:

- `insufficient_capacity` / `insufficient_system_memory` — pick a smaller or more quantized variant, reduce context length, or stop an unused deployment to free budget.
- `vendor_mismatch` / `accel_version_unmet` — the engine does not match your hardware or driver generation; pick a matching variant.
- `insufficient_gpu_count` — lower the tensor-parallel size or free GPUs on a multi-GPU node.
- `sharing_not_configured` — ask your cluster admin to enable GPU sharing.

A SharingNotConfigured notice appears

On a managed cluster where the GPU Operator is installed but no sharing strategy is configured, deployments succeed as whole-GPU and carry a `SharingNotConfigured` notice. This is informational: density is limited to one model per GPU until your cluster admin enables a sharing strategy (such as time-slicing or MIG). The notice clears on its own once sharing is configured.

The deployment reaches ERROR or FAILED after placement

Placement succeeded, but the model failed at runtime. The deployment record carries three fields that tell you what happened:

- **`last_error_code`** — a short classifier such as `OOM`, `CUDA_ERROR`, `MODEL_LOADING_FAILURE`, `CONTAINER_EXITED`, or `STARTUP_TIMEOUT`. The full code list with actions is in Model Deployment (see online documentation).
- **`last_error_message`** — the explanation behind the code: the actual error text captured from the engine or runtime (for example, the CUDA out-of-memory message, or the reason the model files failed to load). Read this first — it usually names the exact resource or file involved, which tells you whether to resize, re-download, or check logs.
- **`last_error_at`** — when the error was recorded, so you can tell a stale error from a current one.

A note on `OOM` after a successful placement: budgets are reservations, not real-time meters. If something outside Kamiwaza consumed GPU or host memory after scheduling, the engine can still run out at load time — see Limits to know (see online documentation). Free memory on the host or reduce the model's footprint, then retry.

The deployment sits in INITIALIZING

Normal for a short period: routing is up but the model is still loading. Large models on unified-memory machines can take a while on first load. If it persists well beyond the expected load time, check `last_error_code` for `STARTUP_TIMEOUT` and see the general troubleshooting guide (see online documentation).

Checking capacity on a standalone cluster

On a standalone cluster you can inspect what placement sees:

```
# GPU labels detected on a node
kubectl get node <node-name> -o jsonpath='{.metadata.labels}' | tr ',' '\n' | grep gpu

# Per-GPU memory budgets advertised on a node (GB)
kubectl get node <node-name> -o jsonpath='{.status.allocatable}' | tr ',' '\n' | grep
vram-gb-gpu
```

If a node shows no `kamiwaza.ai/gpu-*` labels, hardware detection has not labeled it and it will not receive placements.

See also

- Model Placement Overview (see online documentation)
- GPU Hardware Classes (see online documentation)

- Fractional GPU Serving (see online documentation)
- Model Deployment (see online documentation) — lifecycle statuses and error codes
- Model Deployment Troubleshooting (see online documentation) — general issues

AWS Bedrock Model Integration

Overview

Kamiwaza can proxy Amazon Bedrock (https://aws.amazon.com/bedrock/) models through the same OpenAI-compatible APIs used by locally hosted models. Administrators register Bedrock as an external endpoint, deploy it through the normal model lifecycle, and expose familiar `/v1/models` and `/v1/chat/completions` routes to application teams.

The Bedrock integration covers the major model families available in Bedrock — including Anthropic Claude (Sonnet, Haiku, Opus), Amazon Nova, Meta Llama 3, and other Bedrock-hosted families — invoked through the modern Bedrock Converse and InvokeModel APIs.

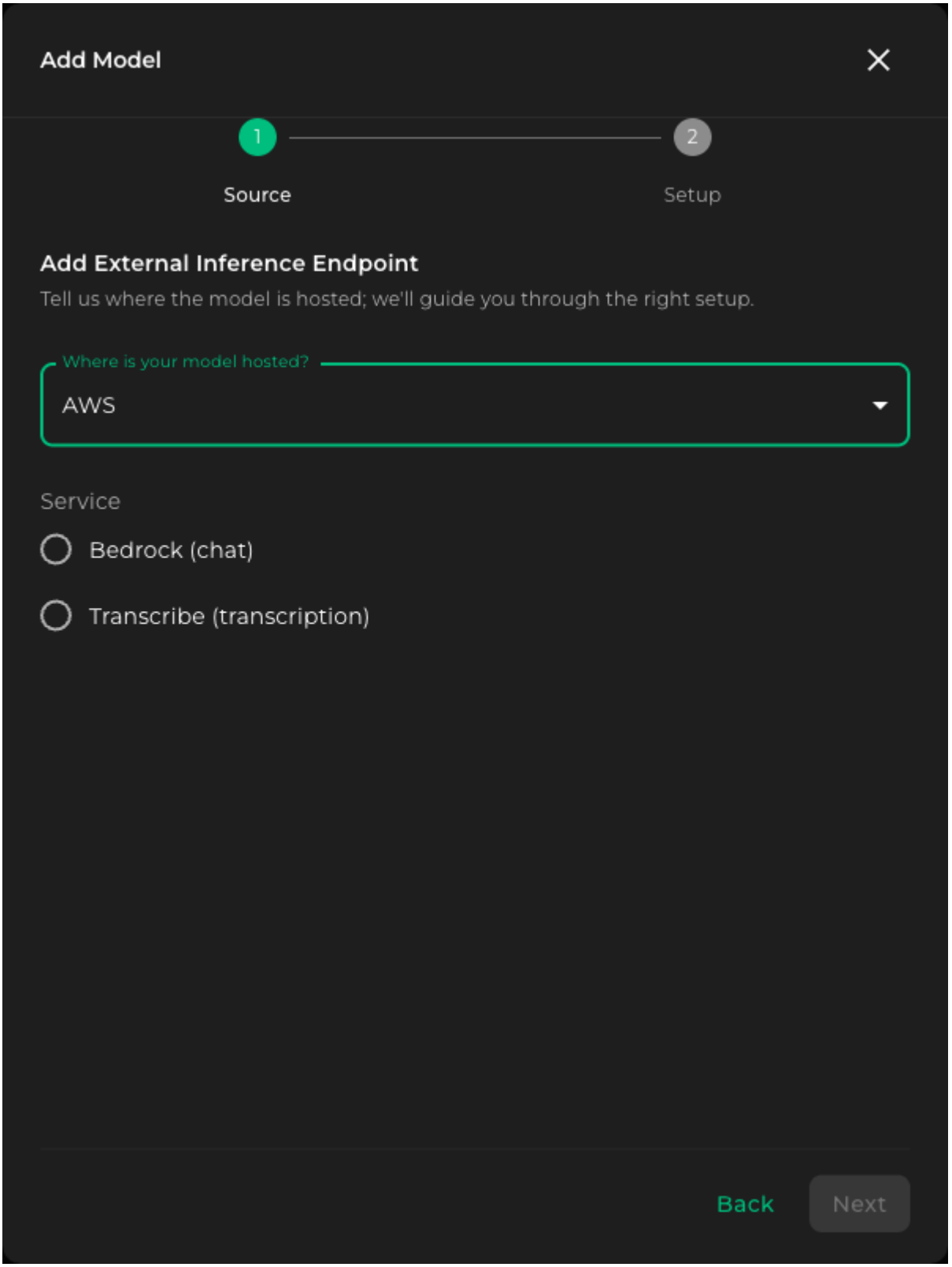
Prerequisites

Before registering a Bedrock endpoint, make sure:

- Your AWS account has Bedrock access enabled in the region you plan to use.
- The target model or inference profile is enabled in that account and region.
- Outbound HTTPS from the Kamiwaza control plane to `bedrock-runtime.<region>.amazonaws.com` is permitted. With an AWS VPC interface endpoint that has private DNS turned on (the AWS default), that hostname already resolves to the endpoint, so nothing else changes. Without private DNS, or for other private connectivity, point the deployment at the private hostname.
- You have either an AWS IAM access key or a Bedrock API key (bearer token) that can invoke the target model.

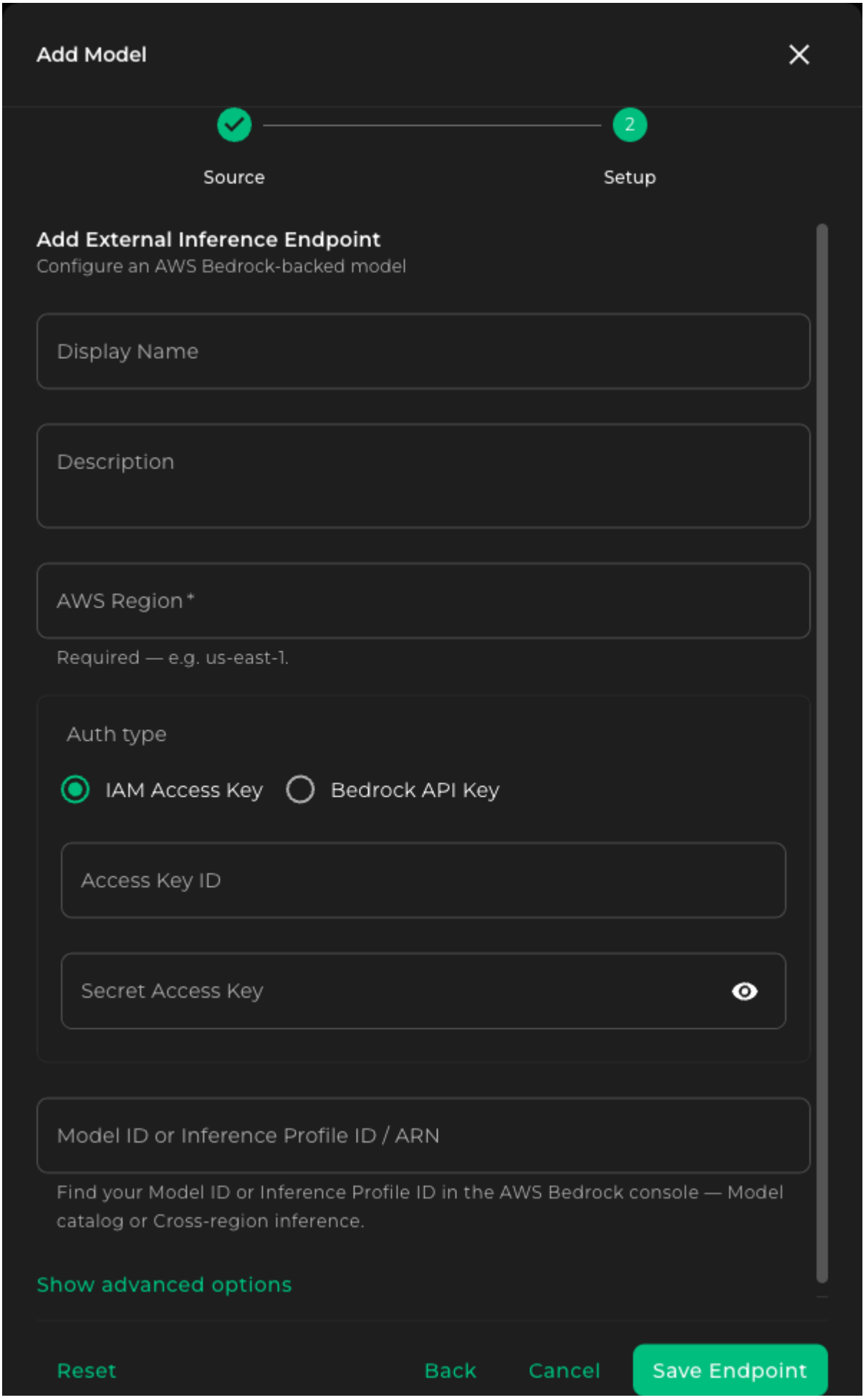
Quick Start (UI)

1. In Kamiwaza, go to **Models**, click **Add Model**, then click **Add external inference endpoint** in the dialog header.
2. On the **Source** step, choose **AWS** under **Where is your model hosted?**, then choose **Bedrock (chat)** under **Service**. Click **Next**.

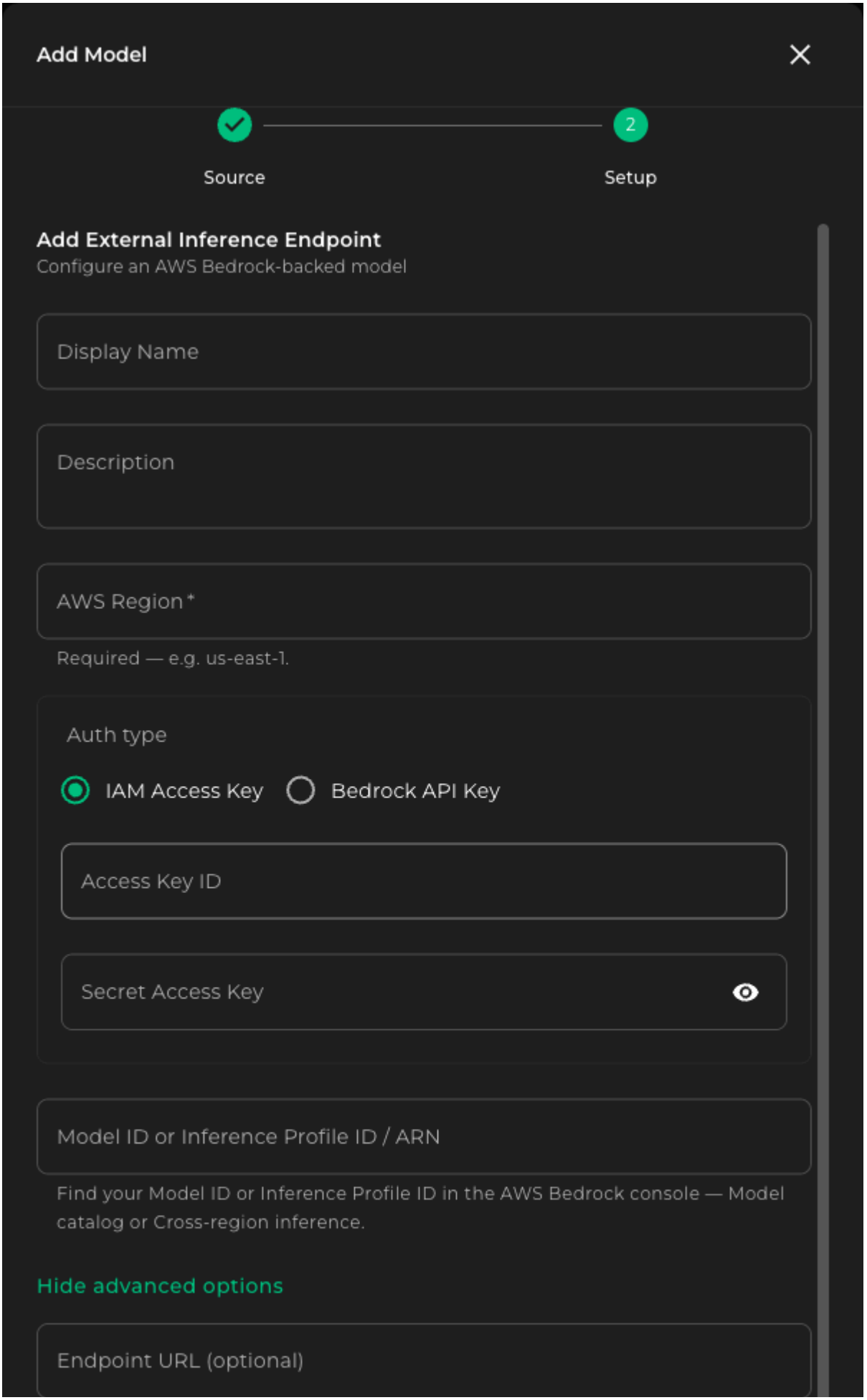


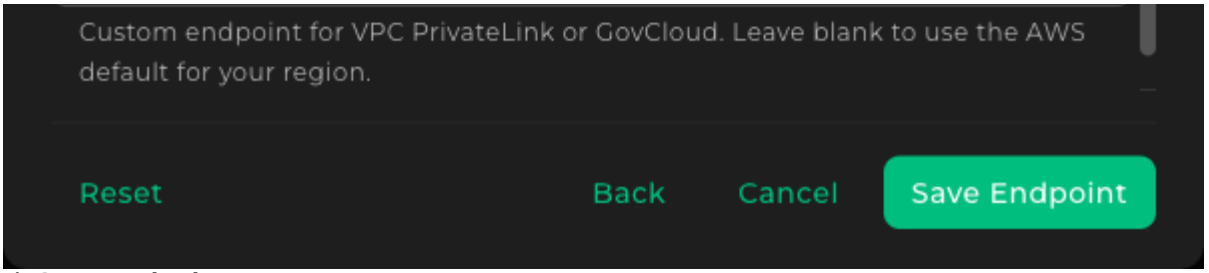
3. On the **Setup** step, fill in the form:
    - **Display Name** — Friendly name shown in the Kamiwaza UI.
    - **Description** *(optional)* — Free-form note for other operators.
    - **AWS Region** — The region where the Bedrock model or inference profile is available (for example `us-east-1`, `us-west-2`, `eu-west-1`). Required.
    - **Auth type** — Choose **IAM Access Key** (the default; long-lived AWS access key + secret) or **Bedrock API Key** (bearer token). Most production setups use IAM access keys.

    - For **IAM Access Key**, paste the **Access Key ID** and **Secret Access Key**.
    - For **Bedrock API Key**, paste the bearer token.
- **Model ID or Inference Profile ID / ARN** — Either a Bedrock model ID, an inference profile ID, or a full ARN. See Model identifiers (see online documentation) below for examples.


Add Model
Source
2
Setup
Add External Inference Endpoint
Configure an AWS Bedrock-backed model
Display Name
Description
AWS Region *
Required — e.g. us-east-1.
Auth type
IAM Access Key
Bedrock API Key
Access Key ID
Secret Access Key
Model ID or Inference Profile ID / ARN
Find your Model ID or Inference Profile ID in the AWS Bedrock console — Model catalog or Cross-region inference.
Show advanced options
Reset
Back
Cancel
Save Endpoint

4. *(Optional)* Click **Show advanced options** to reveal an **Endpoint URL** field. Leave it blank to use the AWS default for your region, including through a VPC interface endpoint with private DNS. Set it explicitly for a VPC endpoint without private DNS, GovCloud, or other non-standard endpoints (placeholder shown is `https://bedrock-runtime.us-east-2.amazonaws.com`).


Add Model
Source
2
Setup
Add External Inference Endpoint
Configure an AWS Bedrock-backed model
Display Name
Description
AWS Region *
Required — e.g. us-east-1.
Auth type
IAM Access Key
Bedrock API Key
Access Key ID
Secret Access Key
Model ID or Inference Profile ID / ARN
Find your Model ID or Inference Profile ID in the AWS Bedrock console — Model catalog or Cross-region inference.
Hide advanced options
Endpoint URL (optional)



5. Click **Save Endpoint**.
6. Deploy the new model from the Models list.

> Use long-lived IAM credentials for production. Temporary or session credentials expire while a deployment is running and lead to silent authentication failures. Kamiwaza no longer accepts a session token field on registration.

Model identifiers

The **Model ID or Inference Profile ID / ARN** field accepts three shapes — pick whichever matches what AWS gave you for the model you want to call.

Shape	Example	When to use
Foundation model ID	`anthropic.claude-sonnet-4-5-20250929-v1:0`	Calling a foundation model directly in a single region.
Inference profile ID	`us.anthropic.claude-sonnet-4-5-20250929-v1:0`	Cross-region inference profiles that AWS published for you.
Inference profile ARN	`arn:aws:bedrock:us-east-1:123456789012:inference-profile/us.anthropic.claude-sonnet-4-5-20250929-v1:0`	Application or custom inference profiles, or any case where you'd rather paste the full ARN.

Use the same field for all three — Kamiwaza recognizes the shape and routes the call accordingly.

Credentials

Credentials registered through the Bedrock form are encrypted on save and stored in the Kamiwaza secret catalog, keyed by AWS region. The catalog rules — credential reuse across endpoints in the same region, rotation through the **Edit** form, and the five-minute propagation window — are described in External endpoints overview (see online documentation).

For **IAM Access Key** authentication, the stored secret is JSON containing your long-lived AWS keys:

```
{
  "aws_access_key_id": "AKIA...",
  "aws_secret_access_key": "..."
}
```

For **Bedrock API Key** authentication, the stored secret is the raw bearer token Bedrock issued. Kamiwaza handles the wrapping when you paste it into the form.

API Usage

Once deployed, the endpoint is callable via the standard Kamiwaza runtime route:

```
curl -X POST "https://<your-domain>/runtime/models/<deployment-id>/v1/chat/completions"
\
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "messages": [
      {"role": "user", "content": "Summarize the Bedrock Converse API in one
sentence."}
    ],
    "max_tokens": 128,
    "temperature": 0
  }'
```

On OpenShift with GreyMatter, the route can take a minute or two to start answering after the deployment shows as deployed. Until then, a call can return `503 no healthy upstream`, or the Kamiwaza web page instead of a JSON response; retry until the response is JSON.

OpenAI-style streaming, tool calls, and structured output are supported on Bedrock model families that support them upstream. Kamiwaza translates between OpenAI wire format and the Bedrock Converse API on each request.

Multimodal input

Claude and Nova deployments accept OpenAI-style multimodal payloads on `/v1/chat/completions`. Image inputs use OpenAI-style content blocks; Kamiwaza converts them to the appropriate Bedrock-native shape.

For Nova, image inputs work with inline `data:` URLs — arbitrary external image URLs are not fetched server-side.

```
{
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "image_url",
          "image_url": {
            "url": "<base64-image-bytes>"
          }
        },
        {"type": "text", "text": "What is shown in this image?"}
      ]
    }
  ],
  "max_tokens": 128,
  "temperature": 0
}
```

Operational notes

- **Region is required.** Bedrock model availability and pricing vary by region; the registration form will not save without one. If you paste a full inference profile ARN, the region in the ARN must match the **AWS Region** field.
- **Model availability is account- and region-specific.** A model ID that works in one AWS account or region may not be enabled in another. Verify in the AWS console before registering.
- Errors from Bedrock — throttling, access denied, model not enabled — are surfaced through the Kamiwaza response body and the audit log.
- For a VPC endpoint without private DNS, GovCloud, or other non-standard endpoints, set the **Endpoint URL** field under **Show advanced options** on the registration form. Ensure the Kamiwaza control-plane network has a permitted path to the chosen hostname before registering.
- **Rotate long-lived IAM credentials regularly.** Treat the registered access key like any other long-lived AWS credential — rotate on the same cadence as the rest of your cloud key inventory and monitor `AccessKeyLastUsed` in IAM. Rotated credentials propagate to running deployments through the **Edit** form within the engine credential cache window (~5 minutes).

Next steps

- Pair the Bedrock deployment with the ReBAC validation checklist to ensure workroom-scoped access controls are enforced.
- Register a transcription endpoint against the same AWS account — see AWS Transcribe (see online documentation).
- Contact Kamiwaza Support if you need help with model enablement, custom inference profiles, or private network connectivity.

AWS Transcribe Engine User Guide

AWS Transcribe (https://aws.amazon.com/transcribe/) provides enterprise-grade, managed speech-to-text capabilities through Kamiwaza's unified transcription API. Audio is processed by AWS's cloud service, eliminating local compute requirements.

Prerequisites

Before registering an AWS Transcribe endpoint, make sure:

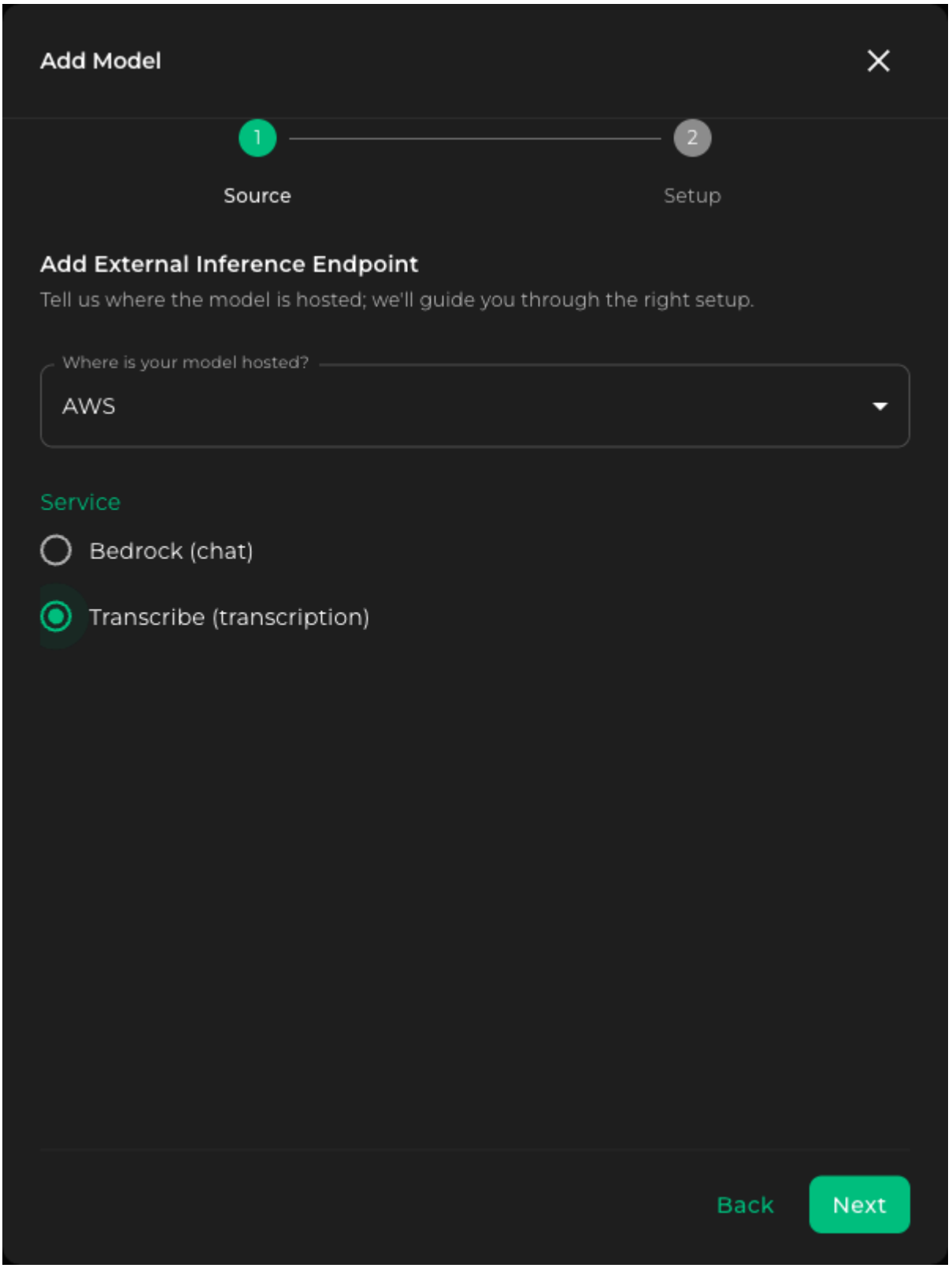
- Your AWS account has Transcribe enabled in the region you plan to use.
- You have an S3 bucket that Kamiwaza can write to for batch transcription (audio is staged there briefly during processing).
- You have a long-lived IAM access key whose attached policy grants the permissions below.
- Outbound HTTPS from the Kamiwaza control plane is permitted to your S3 bucket and to both AWS Transcribe endpoints:
  - `transcribe.<region>.amazonaws.com` — batch transcription jobs.
  - `transcribestreaming.<region>.amazonaws.com` on port 8443 — real-time streaming. Egress-restricted networks must whitelist the streaming hostname separately, otherwise streaming requests will fail to connect.

Minimum IAM policy:

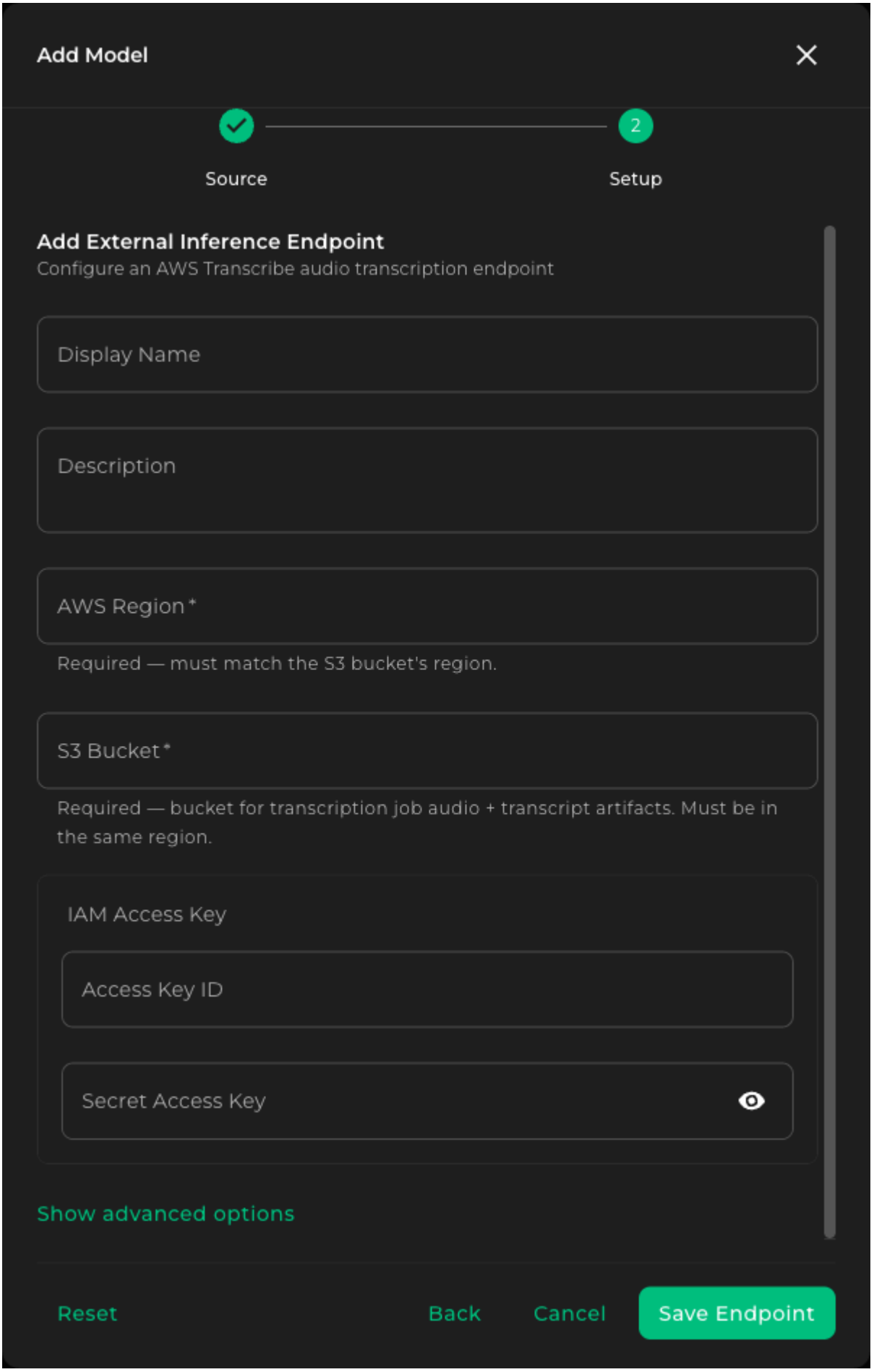
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "transcribe:StartTranscriptionJob",
        "transcribe:GetTranscriptionJob",
        "transcribe:DeleteTranscriptionJob"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject", "s3:GetObject", "s3:DeleteObject"],
      "Resource": "arn:aws:s3:::YOUR-BUCKET/transcribe-jobs/*"
    }
  ]
}
```

Quick Start (UI)

1. In Kamiwaza, go to **Models**, click **Add Model**, then click **Add external inference endpoint** in the dialog header.
2. On the **Source** step, choose **AWS** under **Where is your model hosted?**, then choose **Transcribe (transcription)** under **Service**. Click **Next**.



3. On the **Setup** step, fill in the form:
    - **Display Name** — Friendly name shown in the Kamiwaza UI.
    - **Description** *(optional)* — Free-form note for other operators.
    - **AWS Region** — The region where audio will be processed (for example `us-east-1`). Must match the S3 bucket's region. Required.
    - **S3 Bucket** — The bucket Kamiwaza will use to stage batch audio and transcript artifacts. Must be in the same region. Required.
    - **IAM Access Key** — Paste the **Access Key ID** and **Secret Access Key**.
    - **Show advanced options** *(optional)* — Reveals a **Language** field for setting a default language. Leave it unset to let Transcribe auto-detect the language on each batch request (see Supported Languages (see online documentation)).


Add Model
Source
2
Setup
Add External Inference Endpoint
Configure an AWS Transcribe audio transcription endpoint
Display Name
Description
AWS Region *
Required — must match the S3 bucket's region.
S3 Bucket *
Required — bucket for transcription job audio + transcript artifacts. Must be in the same region.
IAM Access Key
Access Key ID
Secret Access Key
Show advanced options
Reset
Back
Cancel
Save Endpoint

4. Click **Save Endpoint**.
5. Deploy the model from the Models list.

> Use long-lived IAM credentials. Temporary or session credentials expire while a deployment is running and lead to silent authentication failures.

Credentials

Credentials registered through the form are encrypted on save and stored in the Kamiwaza secret catalog, keyed by AWS region. Catalog reuse across endpoints in the same region, rotation through the **Edit** form, and the five-minute propagation window are described in External endpoints overview (see online documentation).

The stored secret is JSON containing your long-lived AWS keys:

```
{
  "aws_access_key_id": "AKIA...",
  "aws_secret_access_key": "..."
}
```

If you have already registered an AWS Bedrock endpoint in the same region **with IAM access-key auth**, Kamiwaza recognizes the existing credential and offers **Use existing credential** during registration. Bedrock endpoints registered with the **Bedrock API Key** auth path store a bearer token, not AWS keys, and cannot be reused for Transcribe — Transcribe requires IAM access keys.

> **Rotate long-lived IAM credentials regularly.** Treat the registered access key like any other long-lived AWS credential — rotate on the same cadence as your cloud key inventory and monitor `AccessKeyLastUsed` in IAM. Rotated credentials propagate to running deployments through the **Edit** form within the engine credential cache window (~5 minutes).

API Usage

Transcribe Audio (Batch)

```
curl -X POST "https://<your-domain>/runtime/models/<deployment-
id>/v1/audio/transcriptions" \
  -H "Authorization: Bearer $TOKEN" \
  -F "file=@recording.wav" \
  -F "response_format=json"
```

Transcribe Audio (Streaming)

```
curl -X POST "https://<your-domain>/runtime/models/<deployment-
id>/v1/audio/transcriptions" \
  -H "Authorization: Bearer $TOKEN" \
  -F "file=@recording.pcm" \
  -F "stream=true"
```

Streaming returns Server-Sent Events (SSE) with partial results.

> **Note:** Streaming only supports `pcm`, `ogg`, and `flac` formats. The format is auto-detected from the file extension. Streaming always uses the registered language (set via **Show advanced options** on the registration form); if none is registered it defaults to `en-US`. The per-request `language` parameter is ignored on streaming.

Request Parameters

Parameter	Required	Description
`file`	Yes	Audio file (multipart form-data)
`response_format`	No	`text`, `json` (default), or `verbose_json`
`language`	No	BCP-47 code. Overrides the registered language on **batch** requests; ignored on streaming requests, which always use the registered language (or `en-US` if none is registered).
`stream`	No	`true` for streaming mode

Response Formats

`json` (default)

```
{
  "text": "Hello, how are you today?"
}
```

`text`

```
Hello, how are you today?
```

`verbose_json`

```
{
  "task": "transcribe",
  "language": "en",
  "duration": 3.45,
  "text": "Hello, how are you today?",
  "segments": [
    { "id": 0, "start": 0.0, "end": 1.2, "text": "Hello," },
    { "id": 1, "start": 1.3, "end": 3.4, "text": "how are you today?" }
  ]
}
```

Batch vs Streaming

Feature	Batch	Streaming
Audio formats	mp3, mp4, wav, flac, ogg, amr, webm	pcm, ogg, flac
Max duration	15 minutes (Kamiwaza default `job_timeout_seconds`; AWS-side hard limit is 4 hours)	5 minutes (300s, AWS-side limit)
Auto language detection	Yes	No (defaults to en-US)
Latency	Higher (job-based)	Real-time
Use case	Recorded audio	Live audio

Supported Languages

AWS Transcribe supports 100+ languages for batch and 30+ for streaming.

Auto-Detect (Batch Only)

Set the **Language** field on the Setup form to **Auto-detect** to let AWS identify the spoken language automatically. This applies to batch mode only.

> **Note:** Automatic language detection is only supported for **batch mode**. For streaming, set the **Language** field on the registration form — streaming uses that registered value (or defaults to `en-US`) and ignores any per-request `language` parameter.

Common Language Codes

Code	Language
`en-US`	English (US)
`en-GB`	English (UK)
`es-ES`	Spanish (Spain)
`es-US`	Spanish (US)
`fr-FR`	French
`de-DE`	German
`ja-JP`	Japanese
`zh-CN`	Chinese (Simplified)
`pt-BR`	Portuguese (Brazil)
`ko-KR`	Korean

Cost Considerations

Check AWS pricing (https://aws.amazon.com/transcribe/pricing/) for current rates.

Troubleshooting

"Unsupported media format"

Batch supports: `mp3`, `mp4`, `wav`, `flac`, `ogg`, `amr`, `webm`, `pcm`. Streaming supports: `pcm`, `ogg`, `flac` only.

"S3 access denied"

Verify the IAM policy attached to the registered access key includes `s3:PutObject`, `s3:GetObject`, and `s3:DeleteObject` on the configured S3 bucket.

"Region mismatch" or 401 / signature errors

Confirm the **AWS Region** on the registered endpoint matches the region your S3 bucket lives in. IAM access keys are global to the account, but Transcribe and S3 are regional services — a region mismatch between the registered endpoint and the bucket surfaces as a 401 or signature error.

Streaming timeout

- Audio chunks must arrive within 30 seconds.
- Total stream duration is limited to 5 minutes (AWS-side limit).
- Ensure audio data is sent promptly from the client.

Job timeout

Kamiwaza enforces a 15-minute (`job_timeout_seconds = 900`) ceiling on each batch job by default. AWS Transcribe itself supports up to 4 hours per job, but the Kamiwaza ceiling will fire first. The form does not currently expose this value, so longer audio must be split into shorter clips or transcribed via the AWS console directly.

OpenAI-Compatible Chat Endpoint Integration

Overview

Kamiwaza can register any external endpoint that speaks the OpenAI `/v1/chat/completions` wire format and proxy it through the same deployment, audit, and access-control surfaces as locally hosted models. This includes OpenAI directly, Microsoft Azure's OpenAI Service and AI Foundry, customer-hosted LiteLLM proxies, self-hosted vLLM or Ollama, and any other provider whose chat API matches the OpenAI shape.

Once registered, the endpoint deploys through the normal model lifecycle and exposes a Kamiwaza `/v1/chat/completions` route that applications can call without knowing which provider sits behind it.

Prerequisites

Before registering an OpenAI-compatible endpoint, make sure:

- You have a provider API key (OpenAI key, Azure resource key, Azure AI Foundry project key, LiteLLM master key, etc.).
- The model you intend to register is enabled on the account the API key belongs to. For Azure, the deployment exists in the project the URL points at.
- Outbound HTTPS from the Kamiwaza control plane to the provider hostname is permitted (see Provider examples (see online documentation) below for hostnames).
- The endpoint URL uses `https://` and resolves to a public address, **or** the install sets `core.scheduler.allowInsecureModelEndpoints: true`. By default, **Connect** refuses a plain `http://` URL with `External endpoints must use HTTPS.` and a private address (for example a vLLM or Ollama server inside your network) with `Base URL must resolve to a public address.` See Plain HTTP and private endpoints (see online documentation).

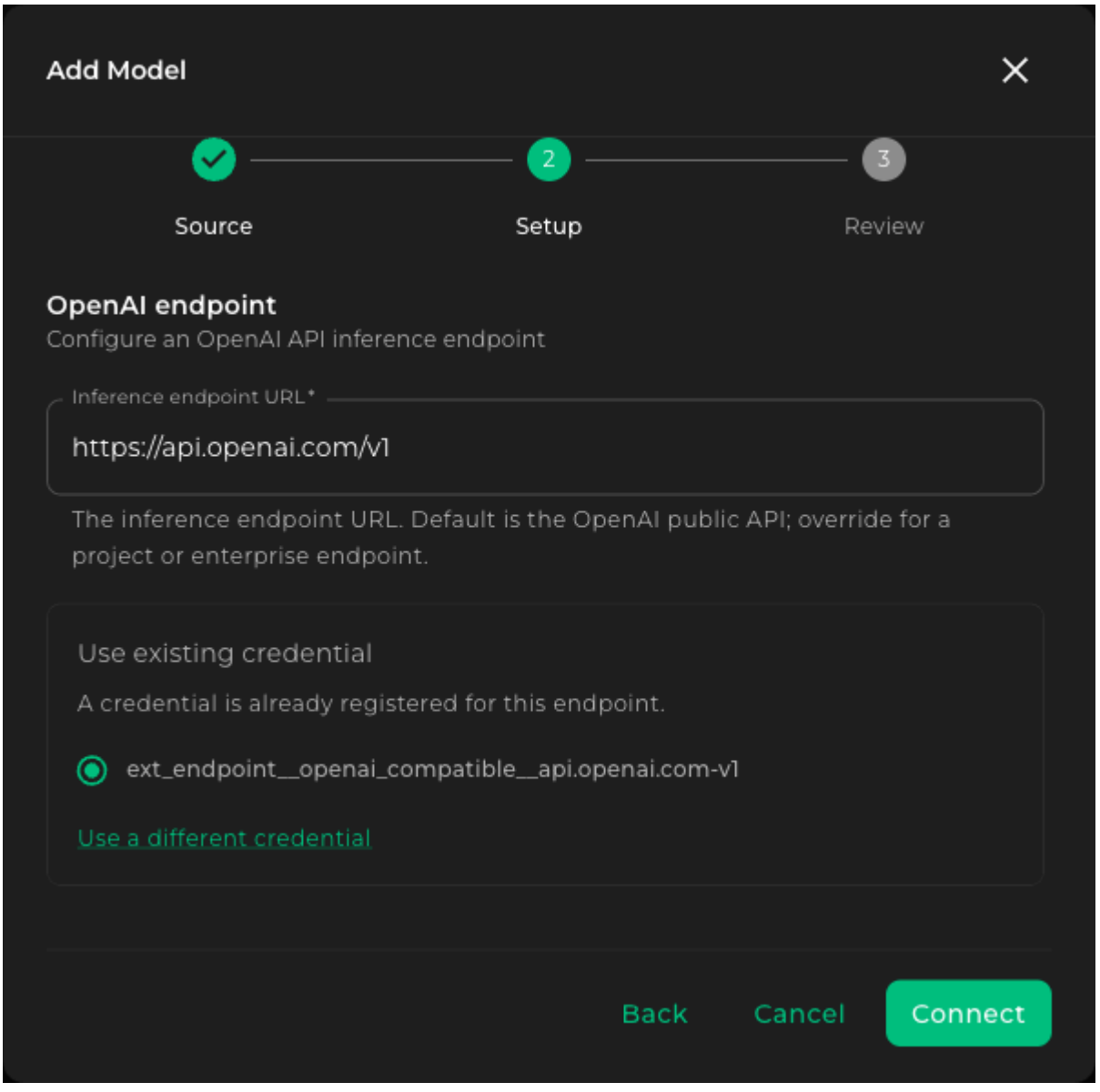
Quick Start (UI)

Registration is a three-step wizard: pick the provider, configure the endpoint, then pick which discovered models to register.

1. In Kamiwaza, go to **Models**, click **Add Model**, then click **Add external inference endpoint** in the dialog header.
2. **Source** — under **Where is your model hosted?**, choose one of:
   - **OpenAI** — for OpenAI directly.
   - **Azure** — for Azure OpenAI Service or Azure AI Foundry. Kamiwaza detects which one from the hostname you paste in step 3.
   - **Other (OpenAI-compatible)** — for LiteLLM proxies, vLLM, Ollama, or any other host whose chat API matches the OpenAI shape.

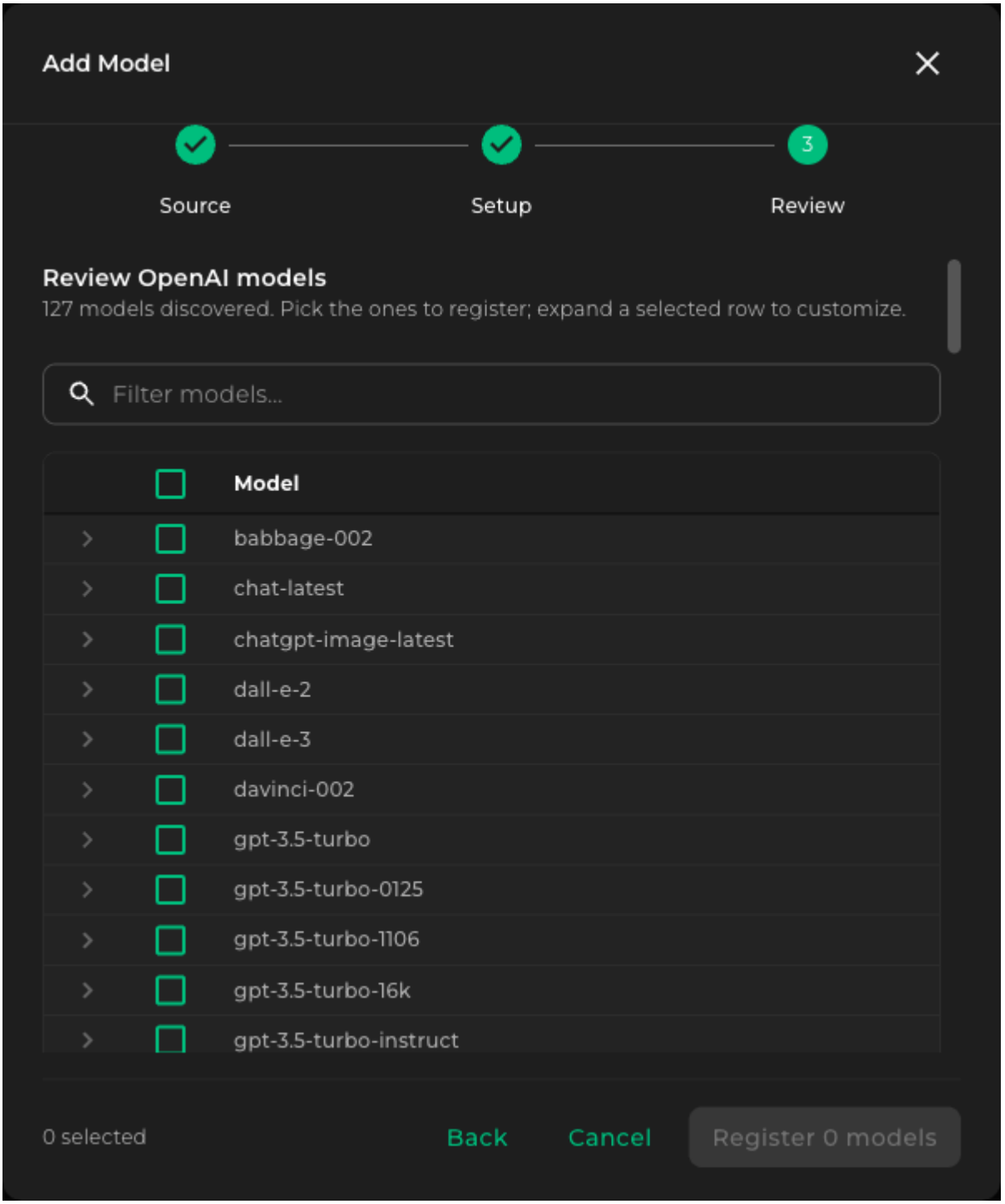
   Click **Next**.

3. **Setup** — configure the endpoint:
    - **Inference endpoint URL** — The provider's API root.
        - For OpenAI: leave the default `https://api.openai.com/v1`. Organization and project scoping are handled by the API key itself (`sk-proj-…` keys) plus optional `OpenAI-Organization` / `OpenAI-Project` headers — there is no separate base URL for them.
        - For Azure: paste the URL Azure gave you (`https://<your-resource>.openai.azure.com` or `https://<your-project>.services.ai.azure.com/models`). Kamiwaza picks the right Azure handler from the hostname.
        - For Other: paste your published proxy or service URL.
    - **Credentials** — If a credential is already stored for this endpoint, Kamiwaza offers **Use existing credential** with a **Use a different credential** link to override. Otherwise paste the provider API key. See Credentials (see online documentation) below for what to use.



Click **Connect**. Kamiwaza validates the URL, authenticates with the credential, and discovers the models the endpoint exposes.

4. **Review** — Kamiwaza shows the list of models discovered on the endpoint. Tick the checkbox for each model you want to register. Expand a row to customize the display name, default parameters, or other per-model settings. Use the **Filter models…** box to narrow the list. Click **Register N models** when done.



5. From the Models list, deploy each newly registered model.

> Test each deployment from the Models list to confirm Kamiwaza can reach the provider with the credentials you supplied before applications start calling it.

Provider examples

The wizard is the same for every provider; only the values change.

OpenAI directly

Field	Value
Source	**OpenAI**
Inference endpoint URL	`https://api.openai.com/v1`
Credential	OpenAI API key (`sk-…`)
Underlying model (Review step)	`gpt-5` , `gpt-4o` , `gpt-5-chat-latest` , etc.

> Restricted OpenAI keys default to chat-only. If you also intend to register a transcription endpoint with the same key, widen the key's scope to include `Audio` first.

Azure OpenAI Service

Field	Value
Source	**Azure**
Inference endpoint URL	`https://<your-resource>.openai.azure.com`
Credential	Azure OpenAI resource key
Underlying model (Review step)	The Azure deployment name (e.g. `gpt-4o`)

Kamiwaza auto-detects the `*.openai.azure.com` host and uses Azure's `api-key` header with a stable `api-version` . You do not need to specify either in the form.

Azure AI Foundry

Field	Value
Source	**Azure**
Inference endpoint URL	`https://<your-project>.services.ai.azure.com/models`
Credential	Azure AI Foundry project key
Underlying model (Review step)	The Foundry deployment name (e.g. `Mistral-Large-3`)

Kamiwaza detects Foundry hostnames separately from Azure OpenAI Service. Azure Government Foundry endpoints (different hostname suffix) are supported through the same handler.

Customer-hosted LiteLLM proxy

Field	Value
Source	**Other (OpenAI-compatible)**
Inference endpoint URL	Your published proxy URL (e.g. `https://litellm.example.com`)
Credential	The proxy's master key (or a virtual key)
Underlying model (Review step)	The model alias as configured in your LiteLLM config

LiteLLM's upstream provider credentials are managed by you on the proxy itself; Kamiwaza only authenticates to the proxy.

Self-hosted vLLM, Ollama, or other OpenAI-compatible service

Use **Other (OpenAI-compatible)**, paste your internal hostname, and supply whatever bearer token the service requires (or any non-empty placeholder if it is open).

A self-hosted server on an internal hostname usually needs the insecure-endpoints setting below, because its address is private and it often serves plain HTTP.

Plain HTTP and private endpoints

Under the default posture, the wizard accepts only an `https://` URL that resolves to a public address. To register a server that serves plain HTTP or sits on a private address, set this value at install time and roll it out with the rest of your Helm values:

```
core:
  scheduler:
    allowInsecureModelEndpoints: true
```

Turn it on only for endpoints on a network you trust: it applies to every endpoint registered on the install, and over plain HTTP the endpoint's API key crosses the network unencrypted. URLs that embed `user:password` credentials, and link-local or cloud-metadata addresses, stay refused even when it is on.

Credentials

Credentials registered through the OpenAI-compatible form are encrypted on save and stored in the Kamiwaza secret catalog, keyed by base URL. Reuse-vs-replace behavior on registration, rotation through the **Edit** form, and the five-minute propagation window are described in External endpoints overview (see online documentation).

For most providers the credential is a single API key string. Kamiwaza handles wrapping it in the right header (`Authorization: Bearer …`, Azure's `api-key`, etc.) based on the detected provider.

Registering and deploying through the API

Scripts can register the same endpoint without the wizard. Send the credential inline as `credential_secret`, inside `external_endpoint`. Kamiwaza encrypts it into the secret catalog and stores only the resulting reference. Do not create the secret first through `POST /api/catalog/secrets/` and pass its reference as `credential_secret_urn`: that reference is scoped to a workroom, and registration refuses it with `403 Referenced credential secret is not readable by requester.`

```
curl -X POST "https://<your-domain>/api/models/" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "my-chat-model",
    "purpose": "chat",
    "default_config": {
      "name": "default",
      "default": true,
      "config": {
        "engine_name": "external_chat",
        "external_endpoint": {
          "protocol": "openai_compatible",
          "base_url": "https://llm.example.com/v1",
          "model_id": "<model id the endpoint serves>",
          "credential_secret": "<API key, or any non-empty placeholder for an open
server>"
        }
      }
    }
  }'
```

The example puts the key on the command line, where shell history keeps it; in practice, write the body to a file readable only by you and send it with `-d @register.json`. Keep to the same URL rules as the wizard (see Prerequisites (see online documentation)). Deploy the registered model with its model ID and default configuration ID from the response, and pass `engine_name`:

```
curl -X POST "https://<your-domain>/api/serving/deploy_model" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"m_id": "<model id>", "m_config_id": "<default_config.id>", "engine_name":
"external_chat"}'
```

In 1.3.4, a deploy request without `engine_name` fails with `500 Internal server error` for an external model.

API Usage

Once deployed, the endpoint is callable via the standard Kamiwaza runtime route:

```
curl -X POST "https://<your-domain>/runtime/models/<deployment-id>/v1/chat/completions"
\
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "messages": [{"role": "user", "content": "Hello"}],
    "max_tokens": 50
  }'
```

Streaming, tool calls, and structured-output requests follow the same shape applications use against any OpenAI-compatible endpoint. Whether the underlying provider supports each capability depends on the specific model.

For reasoning models (OpenAI o-series, gpt-5.x) called through `/v1/chat/completions`, Kamiwaza translates `max_tokens` to `max_completion_tokens` automatically. For OpenAI itself, reasoning controls (`reasoning_effort` and the broader reasoning configuration) belong on OpenAI's Responses API path (`/v1/responses`), where they are forwarded as the `reasoning` object, not on chat completions, where a `reasoning_effort` value is dropped at the param-allowlist boundary.

For a self-hosted server such as vLLM, Ollama, or a LiteLLM proxy, Kamiwaza drops `reasoning_effort` on chat completions unless the model configuration's `external_endpoint` sets `supports_reasoning_effort: true`. With it set, Kamiwaza forwards the requested effort, including `none`, to the server. A value the server rejects returns the server's `400` error. The registration form does not show this setting yet. To set it, read the configuration with `GET /api/model_configs/{model_config_id}`, add `"supports_reasoning_effort": true` inside its existing `external_endpoint` value, keeping that value's other fields and its format, and send the whole configuration back with `PUT /api/model_configs/{model_config_id}`, which replaces it.

Operational notes

- Inference errors from the upstream provider are surfaced through the Kamiwaza response body. Every inference call records requester, workroom, and deployment ID in the Kamiwaza audit log under `event_type=model_invocation`, regardless of upstream success or failure. Use whichever log-aggregation mechanism your deployment provides to surface those entries.
- Multiple registered models that share a base URL share one credential in the catalog. Rotating that credential through any model's **Edit** form updates all of them.
- For environments that require private connectivity (VPC endpoints, private link, mesh-internal proxies), point the **Inference endpoint URL** at the private hostname. A hostname that resolves to a private address is refused unless the install sets `core.scheduler.allowInsecureModelEndpoints: true`; see Plain HTTP and private endpoints (see online documentation). Kamiwaza does not otherwise check that the endpoint is reachable.

Next steps

- Register a transcription endpoint against the same provider — see OpenAI-Compatible Transcribe (see online documentation).
- Pair the deployment with the ReBAC validation checklist to ensure workroom-scoped access controls are enforced.
- Contact Kamiwaza Support for help with non-standard providers, sovereign-cloud hostnames, or private connectivity.

OpenAI-Compatible Transcription Endpoint Integration

Overview

Kamiwaza can register any external endpoint that speaks the OpenAI `/v1/audio/transcriptions` wire format and proxy it through the same deployment, audit, and access-control surfaces as other models. This covers OpenAI's Whisper, Microsoft Azure's Whisper and `gpt-4o-transcribe` deployments, and any self-hosted or third-party Whisper-compatible service.

Once registered, the endpoint exposes the standard `/runtime/models/<id>/v1/audio/transcriptions` route. Kamiwaza's Context Service uses this same route when transcribing audio uploaded by users in workrooms.

Prerequisites

- A provider API key with permission to call the audio-transcription endpoint.
- The transcription model is enabled on the account (or, for Azure, the deployment exists in the project).
- Outbound HTTPS from the Kamiwaza control plane to the provider hostname is permitted (see Provider examples (see online documentation) below for hostnames).

> **OpenAI-specific:** Restricted API keys must be granted the `Audio` scope in addition to chat. Keys created with the default restricted scope set return 401 on transcription calls. Either widen the scope or create a separate key with the `Audio` permission enabled.

Quick Start (UI)

The wizard is identical to the chat-endpoint wizard — only the models you tick on the Review step differ. See OpenAI-Compatible Chat (see online documentation) for screenshots of each step.

1. In Kamiwaza, go to **Models**, click **Add Model**, then click **Add external inference endpoint** in the dialog header.
2. **Source** — choose **OpenAI**, **Azure**, or **Other (OpenAI-compatible)** under **Where is your model hosted?**, then click **Next**.
3. **Setup** — configure the endpoint:
    - **Inference endpoint URL** — The provider's API root. Examples:
        - OpenAI: `https://api.openai.com/v1`
        - Azure OpenAI Service: `https://<your-resource>.openai.azure.com`
        - Azure AI Foundry: `https://<your-project>.services.ai.azure.com/models`
        - Self-hosted Whisper: your internal hostname

- **Credentials** — paste the provider API key, or pick **Use existing credential** if one is already stored for this URL.

Click **Connect**. Kamiwaza validates the URL, authenticates, and discovers the models the endpoint exposes.

4. **Review** — tick the transcription model(s) you want to register (`whisper-1`, `gpt-4o-transcribe`, an Azure deployment name, etc.). Expand a row to customize the display name or per-model defaults. Click **Register N models** when done.

5. Deploy each newly registered model from the Models list.

Provider examples

Provider	Source	Inference endpoint URL	Underlying model
OpenAI Whisper	OpenAI	`https://api.openai.com/v1`	`whisper-1`, `gpt-4o-transcribe`
Azure OpenAI (Whisper, gpt-4o-transcribe)	Azure	`https://<your-resource>.openai.azure.com`	The Azure deployment name
Azure AI Foundry (gpt-4o-mini-transcribe)	Azure	`https://<your-project>.services.ai.azure.com/models`	The Foundry deployment name
Self-hosted Whisper	Other (OpenAI-compatible)	Your internal hostname	The service's published model identifier

If you have already registered a chat endpoint against the same provider URL, Kamiwaza recognizes the existing credential and offers **Use existing credential**. For OpenAI specifically, double-check the key has the `Audio` scope before reusing.

Credentials

Credentials registered through the form are encrypted on save and stored in the Kamiwaza secret catalog, keyed by base URL. Catalog reuse, rotation, and the five-minute propagation window are described in External endpoints overview (see online documentation).

API Usage

```
curl -X POST "https://<your-domain>/runtime/models/<deployment-
id>/v1/audio/transcriptions" \
  -H "Authorization: Bearer $TOKEN" \
  -F "file=@recording.wav" \
  -F "model=whisper-1" \
  -F "response_format=json"
```

Supported request fields: `file` (the audio), `model` (the underlying model), `language` (BCP-47 code), and `response_format`. Other OpenAI Whisper fields (`prompt`, `temperature`) are not forwarded by Kamiwaza on this path. `response_format` accepts `json` (default), `text`, `verbose_json`, `srt`, and `vtt`; values outside that set are normalized to `json`.

Operational notes

- Audio uploads are streamed through Kamiwaza to the upstream provider. Kamiwaza does not retain audio bytes after the response is returned to the caller.
- Transcription errors from the provider are surfaced through the Kamiwaza response body. The requester, workroom, and deployment ID are recorded in the Kamiwaza audit log under `event_type=model_invocation`, regardless of upstream success or failure.
- This path is request/response only — Kamiwaza does not stream OpenAI-compatible transcription. For streaming speech-to-text, register an AWS Transcribe (see online documentation) endpoint instead.

Next steps

- For the streaming and batch capabilities of AWS's managed transcription service, see AWS Transcribe (see online documentation).
- Register a chat endpoint against the same provider — see OpenAI-Compatible Chat (see online documentation).
- Contact Kamiwaza Support for help with non-standard providers, sovereign-cloud hostnames, or private connectivity.

Model Deployment Troubleshooting

This guide helps you diagnose and resolve common issues encountered during model deployment on Kamiwaza.

Common Issues

Deployment Failures

- **Model not found**: Ensure the model exists in your catalog or use Novice Mode to select from the curated list.
- **Checkpoint too large for VRAM**: Choose a smaller/quantized variant (e.g., AWQ, MLX, GGUF) or reduce batch size in Advanced Mode.
- **Service unavailable/port errors**: Retry deployment; if it persists, Stop/Remove and deploy again.
- **Outdated catalog**: Refresh the Models page or restart the server to re-import the model guide.

Performance Issues

- **Slow responses**: Pick a faster model or quantized variant; reduce max tokens and context length.
- **High memory or OOM**: Lower batch size, context length, and KV cache; use a lower-VRAM variant.
- **Cold starts**: First request may be slower while the model loads; send a short warm-up prompt after deploy.

Engine Selection Problems

- **Wrong engine/variant**: Switch to Advanced Mode and explicitly select the engine/variant you want.
- **Mac (MLX) quirks**: Prefer the recommended MLX variant; use the OpenAI endpoint shown in the UI.
- **Task mismatch**: Use coding-tuned models for code, VL models for images, reasoning models for multi-step tasks.

Resource Constraints

- **Insufficient VRAM**: Use a smaller or more heavily quantized model.
- **Low disk space**: Remove unused model files or clear caches; then redeploy.
- **CPU-only environments**: Choose CPU-friendly variants (e.g., GGUF).

Diagnostic Steps

1. Confirm the model status is DEPLOYED in the UI.
2. Open the model’s endpoint URL from the UI; send a short test prompt.
3. If failing, Stop/Remove and redeploy the model.
4. Try a smaller/quantized variant; reduce context length and batch size.

5. For persistent issues, switch to Advanced Mode and review engine/variant settings.

Getting Help

For additional support beyond this troubleshooting guide, see the Need Help? (see online documentation) section for community resources and support channels.

When reporting issues, please share:

- Model name, variant, and engine
- Hardware specs (GPU VRAM/CPU RAM)
- Complete error text or messages
- Steps to reproduce the problem

Model serving 1.3.0 release notes

Kamiwaza 1.3.0 changes four model-serving behaviors on every cluster, including clusters that do not enable tenant-mode inference (see online documentation). No setting turns them on or off. Each one corrects an estimate, a status, or a node label that earlier releases got wrong. The node label needs one step after the upgrade: re-label existing NVIDIA nodes (see online documentation).

Change	Applies to	Visible in
vLLM estimates without a GPU inventory (see online documentation)	vLLM, and estimate requests that name no engine	The VRAM estimate only
KV cache sizing for GGUF models (see online documentation)	GGUF models whose header declares an attention head width	The VRAM estimate and the GPU memory a deployment reserves
A stop clears the deployment's instances (see online documentation)	Deployments stopped after the upgrade	`GET /api/serving/deployment/{deployment_id}`
NVIDIA nodes get the correct CUDA label (see online documentation)	NVIDIA nodes with an R570 or R575 driver (570.x to 579.x), and pre-Turing GPUs with R580 or later	The `kamiwaza.ai/gpu-accel-version` node label and the llama.cpp image a deployment on the node runs

Behavior changes

vLLM estimates without a GPU inventory

Where Kamiwaza finds no GPU in the cluster's node inventory, it now estimates a vLLM deployment's memory as if the deployment will run on an NVIDIA CUDA GPU. Earlier releases estimated it as a CPU deployment, which left out the CUDA graph pool and the GPU runtime's memory. The CPU estimate also assumed flash attention and so counted no attention workspace. Without a GPU inventory, Kamiwaza cannot tell whether the GPU supports flash attention, so the CUDA estimate includes the workspace. vLLM cannot run without a GPU or other accelerator, so the CPU figure was too low. For an 8B model with 16 GiB of weights, the estimate rises from 19.07 GiB to 22.70 GiB at an 8,192-token context, and from 18.48 GiB to 22.12 GiB at a 4,096-token context.

The estimator treats a request that names no engine as vLLM. On such a cluster, `POST /api/serving/estimate_model_vram` (the SDK's `client.serving.estimate_model_vram()`) without `engine_name` therefore also returns the CUDA figure, whichever engine the model would deploy with. The **VRAM** button in a model's **Model Configurations** list sends no engine, so it shows the CUDA figure too, and it offers no way to name one.

This applies to CPU-only clusters, to clusters whose node inventory cannot be read, and to Apple silicon hosts, which run Kamiwaza in a managed Lima virtual machine. A cluster whose inventory reports NVIDIA, AMD, Intel, or Gaudi GPUs is estimated as before. Only the estimate changes: on a cluster with no GPU inventory a deployment reserves no GPU memory, so placement is unaffected.

To compare an estimate with the deployment it describes, pass the deployment's `engine_name` in an API or SDK estimate request.

KV cache sizing for GGUF models

Kamiwaza now reads the attention head width, `<architecture>.attention.key_length`, from a GGUF file's header and sizes the KV cache from it. Earlier releases derived the width by dividing the embedding length by the attention head count, and the KV cache grows in proportion to the width.

The estimate changes wherever the declared width differs from the derived one. It rises where the declared width is larger: Qwen3-4B heads are 128 wide against a derived 80, so earlier releases sized its KV cache at 80/128 of what the model uses. It falls where the declared width is smaller: Mistral-Nemo-Instruct-2407 heads are 128 wide against a derived 160. Where the two agree, as for Qwen3-8B, nothing changes.

Kamiwaza reads the header only when it has no other source for the model's architecture. GGUF models whose architecture Kamiwaza reads from a `config.json` on disk or from Hugging Face, are sized as before, as are GGUF files that do not declare the width. A `head_dim` set in the model configuration still takes precedence over the header.

A higher estimate raises the GPU memory the deployment reserves. On a GPU shared by several models, a deployment that fit before can now fail with a no-fit error (see online documentation). Before 1.3.0 the same deployment reserved less memory than its KV cache needs and could run out of memory when the model loaded. If a deployment no longer fits, reduce its context length, place it on a GPU with more free memory, or stop another deployment on that GPU.

A stop clears the deployment's instances

When a stop succeeds and the deployment reaches `STOPPED`, Kamiwaza now deletes the deployment's instance records. Earlier releases deleted them only when a forced stop had to clean up after a stop that failed. A stop that succeeded left them in place, so a stopped deployment still listed the instance it had while running.

`GET /api/serving/deployment/{deployment_id}` (the SDK's `client.serving.get_deployment()`) now returns an empty `instances` list for a stopped deployment. The response no longer carries the instance's `host_name`, `listen_port`, or `node_id`, so read those fields while the deployment is running.

Deployments stopped before the upgrade keep their instance records until they are deleted.

NVIDIA nodes get the correct CUDA label

The `kamiwaza.ai/gpu-accel-version` label now reads `cuda13` only with an R580 or later driver, and `cuda12` with an older one. A GPU older than Turing (compute capability below 7.5: Maxwell, Pascal, and Volta) has no kernels in a CUDA 13 image, so labeling also caps it at `cuda12` on any driver. The cap gives a Volta GPU (compute capability 7.0) a working llama.cpp image, because the CUDA 12 image carries kernels from compute capability 7.0 up. Maxwell and Pascal GPUs have no llama.cpp CUDA image in 1.3.0, whatever the label says.

Earlier releases labeled an NVIDIA node `cuda13` from R570 up, with no GPU cap. A node with an R570 or R575 driver (570.x to 579.x) therefore received a CUDA 13 llama.cpp image that fails with "CUDA driver version is insufficient", and a pre-Turing GPU with R580 received one with no kernels for the GPU. The label changes on those nodes only; a node with an older driver, such as R535 or R550, was already `cuda12`. A node keeps its old label until it is labeled again, so correct it after the upgrade, as described in Re-label existing NVIDIA nodes (see online documentation).

On a node that the NVIDIA GPU Operator labels and that carries no `kamiwaza.ai/gpu-vendor` label, Kamiwaza derives the CUDA generation from the operator's labels each time it deploys, with the same compute-capability cap. Such a node needs no re-label.

The label selects the llama.cpp image for a deployment on the standard deployment path, where Kamiwaza can read the cluster's Nodes. An install that does not grant Kamiwaza read access to Nodes cannot see the label, so the label does not select the image there. A tenant-mode deployment does not read node labels: it selects an image from the facts in the owner's signed profile, as described in tenant-mode inference (see online documentation).

A Kamiwaza chart install with default values does not grant read access to Nodes. To check whether your install has it, run the following as a cluster administrator, with `<NAMESPACE>` set to the namespace Kamiwaza is installed in:

```
for sa in core-api core-ray core-scheduler; do
  kubectl auth can-i list nodes --as="system:serviceaccount:<NAMESPACE>:${sa}"
done
```

If every line prints `yes`, Kamiwaza can read Nodes, and the label selects the llama.cpp image on the standard deployment path. If they print `no`, Kamiwaza cannot read Nodes, and the label does not select the image.

Re-label existing NVIDIA nodes

Tenant-mode installs. No re-label is needed: tenant-mode inference does not read node labels. The driver's CUDA major that selects the image comes from the owner's signed profile. After a driver change, re-detect the node's facts and publish a new signed profile revision; see Attest the NVIDIA driver's CUDA major (see online documentation).

Installs that grant Kamiwaza read access to Nodes. Set the label on each NVIDIA node that carries `kamiwaza.ai/gpu-vendor`. On the node, read the driver version and each GPU's compute capability:

```
nvidia-smi --query-gpu=driver_version,compute_cap --format=csv,noheader
```

```
570.172.08, 8.0
```

The label is `cuda13` when the driver is 580 or later and every GPU's compute capability is 7.5 or later, and `cuda12` otherwise. The node above, with an R570 driver, is `cuda12`. Set it, with `<CUDA_LABEL>` as `cuda12` or `cuda13`:

```
kubectl label node <NODE_NAME> kamiwaza.ai/gpu-accel-version=<CUDA_LABEL> --overwrite
```

Confirm the new value on every NVIDIA node:

```
kubectl get nodes -L kamiwaza.ai/gpu-accel-version
```

A manual label does not help a Maxwell or Pascal GPU (compute capability below 7.0): no llama.cpp CUDA image in this release has kernels for it.

A running deployment keeps the image it started with. Redeploy a llama.cpp model on a re-labeled node to move it to the image the new label selects.

Keep `core.gpu.nodeFeatureRule.enabled` at its default, `false`: when it is `true` on a cluster with Node Feature Discovery installed, the chart's NodeFeatureRule writes `sm_XX` values such as `sm_80` to `kamiwaza.ai/gpu-accel-version` on the GPU models it lists, replacing `cuda12` and `cuda13`.

App Garden

App Garden lets you browse, deploy, and manage curated applications from the Kamiwaza UI. Applications are launched through the platform's managed deployment flow and exposed through the same authenticated routing model used elsewhere in Kamiwaza.

What is App Garden?

App Garden is a catalog of ready-to-run apps (dashboards, demo UIs, tools) that you can deploy in your Kamiwaza environment. It handles all the technical setup for you, so you can focus on using the app.

For documentation on specific apps available in the App Garden, please check the Extensions (see online documentation) section. If you want to build and ship your own app, service, or tool for Kamiwaza, see the Developer Guide (see online documentation).

Key Features

- **One-click deploy**: Launch apps directly from the catalog
- **Automatic routing**: Each app gets a stable HTTPS URL through the platform gateway
- **Browser-based access**: Users can open deployed apps directly from the Kamiwaza UI
- **AI-ready**: Apps can automatically connect to your deployed models (OpenAI-compatible)
- **Simple lifecycle**: Start, stop, and remove from the UI
- **Remote catalog sync**: Pull templates from the remote catalog with version filtering
- **Template variables**: Environment values can reference routing-aware variables

Getting Started

1. Open the App Garden page in the Kamiwaza UI.
2. If the catalog is empty, click Import/Refresh (or ask your administrator to enable the remote catalog).
3. Browse the list and select an app to view details.
4. Click Deploy. App Garden will start the containers and assign a URL.
5. Click Open to launch the app in your browser.

Deploying and Managing Apps

- **Deploy**: Choose an app and click Deploy. Most apps work out of the box with defaults.
- **Access**: After deployment, use the Open button or copy the provided URL.
- **Status**: Check deployment status, ports, and health in the App Garden page.
- **Stop/Remove**: Stop or remove an app anytime from its details panel.

Using AI Models with Apps

Many apps can use models you have deployed in Kamiwaza. App Garden provides standard OpenAI-compatible environment variables to the app automatically, so most apps need no manual configuration.

Tips:

- If your app has a model preference setting (e.g., fast, large, reasoning, vision), choose it in the app's configuration panel before deploying.
- Ensure at least one model is deployed if your app requires AI.

Remote template catalog

App Garden templates can sync from a remote catalog. Administrators can choose the approved catalog source for their environment and refresh the catalog when new templates are published.

Template variable substitution

Template environment variables can reference routing-aware values. Examples include:

- `{openai_base_url}`: the OpenAI-compatible base URL selected by the platform for the model, typically path-based in current deployments (no trailing `/v1`)
- `{openai_path_base_url}`: the explicit path-based OpenAI base URL (no trailing `/v1`)
- `{model_path_url}`: full HTTPS URL to the selected model (path-based)
- `{app_path_url}`: full HTTPS URL to the app (path-based)

Reserved `KAMIWAZA_*` and `FORWARDAUTH_SIGNATURE_SECRET` environment keys are protected and cannot be overridden by user input.

When to Use App Garden

- You want a quick, reliable way to run common tools and demos
- You prefer a click-to-deploy experience over manual Docker commands
- You need apps that "just work" with your existing model deployments

Troubleshooting

- **No apps in the catalog**: Click Import/Refresh on the App Garden page, then retry. If still empty, ask your administrator to enable the default catalog.
- **App will not start**: Retry Deploy. If it persists, Stop/Remove and deploy again.
- **Cannot reach the app**: Use the Open button from the UI. Avoid direct container ports; App Garden routes traffic for you.

- **AI features not working**: Verify at least one model is deployed and healthy. Some apps expose a preference for model type—set it before deployment.

Ephemeral Sessions

When deploying an app, you can enable **Ephemeral session** mode. This automatically cleans up the deployment when you log out or your session expires.

- **Check the box** during deployment to make the app ephemeral
- **Ephemeral apps** are automatically purged on logout or session timeout
- **Persistent apps** (unchecked) remain running until manually stopped

This is useful for demo environments or when you want automatic cleanup of test deployments.

> **ADMINISTRATOR CONFIGURATION**
>
> Administrators can force all deployments to be ephemeral by setting `KAMIWAZA_EPHEMERAL_EXTENSIONS=true`. See the Administrator Guide (see online documentation) for details.

Session tokens for apps

When an app is deployed in ephemeral mode, it receives a session token and endpoints it can call to keep the session alive or request cleanup:

- `KAMIWAZA_APP_SESSION_TOKEN`
- `KAMIWAZA_APP_SESSION_HEARTBEAT_ENDPOINT` (`/api/apps/sessions/heartbeat`)
- `KAMIWAZA_APP_SESSION_ENDPOINT` (`/api/apps/sessions/end`)

Apps should send periodic heartbeats when they remain active. Ending the session (or logging out) triggers cleanup of the associated deployment.

Advanced Options

App customization is coming soon!

Tool Shed

Tool Shed lets you deploy and manage tool servers from supported container images or curated templates. It is designed for tool-enabled applications and agent workflows that need dedicated, platform-managed endpoints.

Kamiwaza typically leverages Model Context Protocol (MCP) servers to provide tools to agentic workflows via these.

For documentation on specific tools available in the Tool Shed, please check the Extensions (see online documentation) section. If you want to build and ship your own app, service, or tool for Kamiwaza, see the Developer Guide (see online documentation).

What you can do

- Deploy a tool server from a container image
- Deploy from a template in the remote catalog
- List, inspect, and stop tool deployments
- Reuse the same remote template catalog used by App Garden

Deploy from a template

1. Open **Tool Shed** in the Kamiwaza UI.
2. Choose a tool template.
3. Provide required environment variables (for example API keys).
4. Click **Deploy**. A public HTTPS URL is generated for the tool server.

Remote template catalog

Tool templates can sync from a remote catalog (the same catalog used by App Garden). Administrators can:

- Choose the catalog stage (LOCAL, DEV, STAGE, PROD)
- Refresh templates on demand
- See which templates are new or updated

This enables rapid rollout of new tools without shipping static JSON files.

API highlights

These endpoints are available behind the standard API gateway:

- `POST /api/tool/deploy` (deploy from image)

- `POST /api/tool/deploy-template/{template_name}` (deploy from template)
- `GET /api/tool/deployments` (list deployments)
- `DELETE /api/tool/deployment/{deployment_id}` (stop and remove)

Notes

- Tool deployments get stable HTTPS URLs and integrate with the same routing system as models and apps.
- Use secrets from the Data Catalog for any sensitive configuration values.

Distributed Data Engine

Kamiwaza's Distributed Data Engine (DDE) aligns unstructured and tabular content under a unified ingestion framework. Connector-driven pipelines push cleaned documents into the platform's vector stores while keeping credentials and scheduling consistent with the rest of the control plane. Subsequent point releases continue to build on the same structure, so the guidance below applies to the current GA build unless otherwise noted.

> DDE stands for **Distributed Data Engine**.

Platform assumptions

- The current Kamiwaza platform is deployed and available over HTTPS.
- You have administrative access to configure connectors and manage secrets.
- Target storage (Kamiwaza vector database or an external store) is reachable from the ingestion workers.
- The platform uses a dedicated, standalone OpenAI-compatible `llama.cpp` embedding service for vector generation, deployed via Helm.

Connector workflow

1. **Create a connector** – Choose the source system (S3, SharePoint, file upload, and so on) and supply the required fields through the UI or supported API.
2. **Secure credentials** – Prefer referencing an existing Kamiwaza secret. Secrets are encrypted at rest and can be rotated without recreating the connector.
3. **Schedule ingestion** – Select one-time or recurring runs. DDE batches updates to minimize load on the source system.
4. **Monitor jobs** – Each run emits status events and logging metadata that flow into the standard Kamiwaza logging and observability paths.

Supported sources

Source	Notes
File	Files accessible to the deployment through supported upload or storage paths.
Amazon S3	Uses access key/secret with read permission on the target bucket/prefix.
Kafka	Streams and batch pulls for topic-backed ingestion.
Postgres	Reads structured data for catalog and retrieval.
Hive	Reads warehouse data for catalog and retrieval.
Slack	Pulls channel content for indexing and retrieval.

> Need a connector that is not listed? Contact Kamiwaza Support to discuss roadmap status or professional-services extensions.

Ingestion service APIs (scheduling + runs)

The ingestion service provides job-based ingestion and connector execution. These endpoints are available behind the standard API gateway:

- `POST /api/ingestion/ingest/run` – run a connector immediately
- `POST /api/ingestion/ingest/jobs` – schedule a connector run (cron syntax)
- `GET /api/ingestion/ingest/status/{job_id}` – check job status

Each ingestion request specifies a `source_type` (for example `s3` or `postgres`) and a `kwargs` / `conn_args` payload that holds the connector-specific configuration.

DDE connector and document APIs

DDE connector and document endpoints are mounted under the ingestion service with `/api/dde/...` paths. These are used by the UI and by automated ingestion workflows:

- `POST /api/ingestion/api/dde/connectors` – create a connector
- `PATCH /api/ingestion/api/dde/connectors/{id}` – update a connector
- `POST /api/ingestion/api/dde/connectors/{id}/trigger_ingest` – run now
- `POST /api/ingestion/api/dde/documents` – index a document
- `GET /api/ingestion/api/dde/documents` – list documents for a connector

Connectors carry security metadata such as `system_high` (the maximum classification allowed) and an optional `default_security_marking` applied when documents lack explicit markings.

Audio ingest

Audio files submitted through the context manager are routed to the platform's OmniParse transcription endpoint before indexing. The context manager's supported media extensions are `.aac`, `.aif`, `.aiff`, `.flac`, `.m4a`, `.m4b`, `.mp3`, `.oga`, `.ogg`, `.opus`, `.wav`, `.wma`, plus `.mov`, `.mp4`, and `.webm` when only the audio track needs to be transcribed.

Transcribed text enters the same extraction and indexing path as documents, so media sources appear in search and retrieval with the same security markings as their parent connector. No connector-side configuration change is required; the routing is automatic based on file type when OmniParse is configured. The automatic OmniParse path enables `use_omniparse=true` with `strict_omniparse=false` in the underlying context manager. That means OmniParse failures fall back to the built-in extractors where one exists, instead of failing the whole job immediately. For audio/video files themselves, successful transcription is still required to produce chunks, so empty transcription results behave like empty content.

Current limits come from the context manager rather than a DDE-specific knob: the default decoded file-size limit is `100 MB` (configurable via `CONTEXT_MANAGER_MAX_FILE_SIZE`), the streaming upload ceiling is `200 MB`, and the OmniParse transcription timeout defaults to `300` seconds.

For detailed documentation on OmniParse deployment, capabilities, and configuration options, see the OmniParse Extension Guide (see online documentation).

DDE MCP tool

Kamiwaza also ships a DDE-focused MCP tool, `tool-kamiwaza-dde`, for apps and agents that need to work with catalog, ingestion, retrieval, VectorDB, or knowledge-graph operations without building their own REST clients.

Area	Example tools	What they are used for
Catalog	`search_catalog`, `get_dataset`, `add_to_catalog`	Discover dataset metadata and register datasets in the catalog.
Pipelines	`ingest_source`, `ingest_document`, `create_pipeline_job`, `get_pipeline_job`	Submit content and manage ingestion jobs.
Retrieval	`search_context`, `retrieve_context`, `agentic_search`	Run grounded context search and retrieval from agents.
VectorDB	`list_collections`, `search_collections`, `query_vectors`	Inspect and query retrieval collections and vector stores.
Knowledge graph	`add_knowledge`, `search_knowledge`, `get_memory`	Populate and query Graphiti-backed knowledge state.

The exact tool inventory can vary by release. The current line aligns the MCP surface more closely to the REST APIs exposed by the platform. To inspect the exact tool IDs your deployment exposes, send the standard MCP `tools/list` request after `initialize`:

```
{
  "jsonrpc": "2.0",
  "id": "2",
  "method": "tools/list",
  "params": {}
}
```

Session and auth flow

The DDE MCP tool uses streamable HTTP MCP semantics:

1. Send `initialize` to `/mcp`.
2. Read the returned `MCP-Session-Id` and `MCP-Protocol-Version` headers.
3. Send both headers on subsequent requests in the same session.

Operational notes:

- Restrict browser access with `KAMIWAZA_ALLOWED_ORIGINS`. The DDE MCP tool reads this variable directly; it does not use the platform-level `KAMIWAZA_CORS_ORIGINS` setting.
- For static service-to-service auth, set `KAMIWAZA_API_TOKEN`. If that variable is unset, the tool falls back to `KAMIWAZA_API_KEY`.
- If the incoming MCP request already carries `Authorization` or a forwarded `access_token` cookie, the tool preserves that caller/session auth instead of overriding it with the static env token.
- For end-user flows that should preserve the caller's identity, install the shared auth bridge so forwarded Kamiwaza headers reach the tool and downstream APIs.

Security markings and rate limits

DDE document indexing and retrieval enforce security markings and system-high rules:

- Requests should include the user's system-high clearance via the `X-User-System-High` header.
- Document markings are validated against system-high before indexing or listing.
- Requests are rate limited per connector and requester (HTTP 429 with `Retry-After` on limit).

Default rate limits are controlled by `DDE_DOCUMENT_RATE_LIMIT` (requests per window) and `DDE_DOCUMENT_RATE_WINDOW_SECONDS`.

Operational guidance

- Keep connector credentials scoped to read-only roles where possible.
- Use the ReBAC validation checklist to ensure only authorized administrators can create or run connectors.
- Combine DDE jobs with Kamiwaza's retrieval pipelines to expose fresh content in RAG applications.
- For connector-specific tuning such as chunking and file-size limits, use the supported product configuration options and deployment guidance available for your environment.

Data Connectors

Data connectors in Kamiwaza allow administrators to configure external data source connectors. Once configured, users can connect their accounts to access data from these services, enabling integration with Microsoft 365, Google Workspace, and other cloud services.

Overview

External data connectors enable users to access their data from cloud services like Microsoft 365 and Google Workspace. Administrators configure the connector settings (such as Azure AD application registration for Microsoft 365), and then users can connect their personal or work accounts to access data from these services.

The connector configuration includes:

- **Service configuration**: Application credentials and settings required to connect to the external service
- **Access controls**: Permissions and scopes that determine what data users can access
- **Status tracking**: Whether the connector is configured and ready for users to connect their accounts

Connectors are managed through the Kamiwaza web UI at **Settings → Data Connectors**.

Configuring a Connector

Prerequisites

- Administrative access to the Kamiwaza platform
- Application credentials for the external service (e.g., Azure AD app registration for Microsoft 365)
- Knowledge of the service's authentication and API requirements

Using the Web UI

1. Navigate to **Settings → Data Connectors**
2. You will see available connector types:
    - **Microsoft 365** - Connect to SharePoint, OneDrive, Outlook, and Calendar
    - **Google Workspace** - Connect to Drive, Gmail, and Calendar (read-only)
    - **Dropbox** - Connect to Dropbox storage
3. Click **Configure** on the connector you want to set up
4. Fill in the required configuration details:
    - For Microsoft 365: Azure AD application credentials (Client ID, Client Secret, Tenant ID)
    - For Google Workspace: Google Cloud OAuth credentials (Client ID, Client Secret, Redirect URI) plus the read-only capabilities (Gmail, Drive, Calendar) to request — see Google Workspace Connector (see online documentation) for the full walkthrough
    - Service-specific settings and permissions
5. Click **Save** to complete the configuration

Once configured, the connector status will show as "Configured" and users will be able to connect their accounts to access data from these services.

Managing Connectors

Viewing Connectors

In the **Settings** → **Data Connectors** page, you can view all available connector types. Each connector card shows:

- Connector name and logo
- Associated services (e.g., "SharePoint, OneDrive, Outlook, Calendar" for Microsoft 365)
- Configuration status ("Configured" or "Not configured")
- Availability status (some connectors may show "Coming Soon")

Viewing Connector Configuration

Click on a configured connector to view its configuration details, including:

- Application credentials and settings (sensitive values are redacted)
- Configured permissions and scopes
- Connection status

Updating a Connector

To update a connector's configuration:

1. Navigate to **Settings** → **Data Connectors**
2. Click **Configure** on the connector you want to update
3. Modify the configuration fields as needed:
    - Application credentials (Client ID, Client Secret, etc.)
    - Service-specific settings
    - Permissions and scopes
4. Click **Save** to apply changes

Note: Updating connector configuration may require users to reconnect their accounts if authentication settings change.

Removing a Connector Configuration

To remove a connector configuration:

1. Navigate to **Settings** → **Data Connectors**
2. Click **Configure** on the connector
3. Click **Delete** to remove the configuration

Warning: Removing a connector configuration will disconnect all users who have connected their accounts. Users will need to reconnect after the connector is reconfigured.

User Connection Flow

Once an administrator has configured a connector, users can connect their accounts:

1. Users navigate to their profile or data sources section
2. They see available connectors that have been configured by administrators
3. Users click to connect their account (e.g., "Connect Microsoft 365")
4. They are redirected to authenticate with the external service (OAuth flow)
5. After successful authentication, their account is connected and they can access data from that service

The connector configuration determines what permissions and data scopes are available to users when they connect their accounts.

Security and Access Control

Application Credentials

- **Secure Storage**: Application credentials (Client IDs, Client Secrets) are encrypted at rest
- **Credential Rotation**: Update the connector configuration in the UI to rotate application credentials when needed
- **Least Privilege**: Configure application permissions to request only the minimum scopes needed

User Access

- **OAuth Authentication**: Users authenticate directly with the external service (Microsoft, Google, etc.) using OAuth
- **User-Level Permissions**: Each user's access is limited to their own account and the permissions they grant during the OAuth flow
- **Admin Control**: Administrators control which connectors are available to users, but users control which accounts they connect

Data Access

- **Scoped Access**: Users can only access data from accounts they have connected
- **Permission Boundaries**: The connector configuration determines what permissions are requested from users during the OAuth flow
- **Data Isolation**: Each user's connected accounts and data are isolated from other users

Connector Status

Each connector displays its configuration status:

- **Not configured**: The connector has not been set up yet. Administrators need to click **Configure** to set up the connector.
- **Configured**: The connector is ready for users to connect their accounts. Users will see this connector as available in their data sources.

View connector status on the **Settings** → **Data Connectors** page. The status indicator shows whether each connector type is configured and ready for use.

Supported Connector Types

Connector	Services	Status
Microsoft 365	SharePoint, OneDrive, Outlook, Calendar	Available
Google Workspace	Drive, Gmail, Calendar (read-only)	Available
Dropbox	Dropbox storage	Coming Soon

Microsoft 365 Connector

The Microsoft 365 connector allows users to connect their Microsoft accounts to access:

- **SharePoint**: Document libraries and sites
- **OneDrive**: Personal file storage
- **Outlook**: Email messages
- **Calendar**: Calendar events and meetings

Configuration Requirements:

- Azure AD application registration
- Client ID and Client Secret
- Tenant ID (for single-tenant apps)
- Required API permissions configured in Azure AD

Google Workspace Connector

The Google Workspace connector lets users connect their Google accounts to access, **read-only**:

- **Google Drive**: files and folders
- **Gmail**: email messages
- **Google Calendar**: calendar events

Administrators register a Google Cloud OAuth client once; each user then connects their own Google account through Google's consent screen. The connector requests **read-only** scopes only — it cannot send mail, modify files, or change calendar events.

Configuration Requirements:

- A Google Cloud project
- An **OAuth 2.0 Client ID** of type **Web application** (provides the Client ID and Client Secret)
- The required Google APIs **enabled** in that project: **Gmail API**, **Google Drive API**, **Google Calendar API**
- An **Authorized redirect URI** that exactly matches Kamiwaza's callback (see Step 2)

- A configured **OAuth consent screen** (the requested scopes are *sensitive/restricted*, so external apps may require Google verification — see the note below)

Step 1 — Create the OAuth client in Google Cloud Console

1. Open the Google Cloud Console (https://console.cloud.google.com/) and create or select a project.
2. **Enable the APIs** under **APIs & Services → Library** — enable each capability you plan to offer:
   - **Gmail API**, **Google Drive API**, **Google Calendar API**
   - *(If an API is not enabled, that capability fails permanently and affected users are told to reconnect after the admin enables it.)*
3. Configure the **OAuth consent screen** (**APIs & Services → OAuth consent screen**):
   - Choose **Internal** (recommended for a single Google Workspace organization) or **External** (requires adding test users, or publishing the app + Google verification).
   - Add the scopes you will request: `gmail.readonly`, `drive.readonly`, `calendar.readonly`, plus the basic `userinfo.email` and `userinfo.profile` sign-in scopes.
4. Create the credential under **APIs & Services → Credentials → Create credentials → OAuth client ID**:
   - **Application type: Web application.**
   - Under **Authorized redirect URIs**, add Kamiwaza's callback URL (exact match — see Step 2).
5. Copy the **Client ID** (ends with `.apps.googleusercontent.com`) and the **Client Secret** (starts with `GOCSPX-`).

Step 2 — Configure the connector in Kamiwaza

1. Go to **Settings → Data Connectors** and click **Configure** on **Google Workspace**.
2. Fill in:
   - **Display Name** — a friendly label for the connector.
   - **Client ID** and **Client Secret** — from Step 1.
   - **Redirect URI** — prefilled from your current browser origin as `https://<host>/api/connectors/public/google/callback`. The path `/api/connectors/public/google/callback` is fixed; set the **scheme and host** to the address your end users will use. **This value must exactly match an Authorized redirect URI in your Google Cloud OAuth client** (scheme, host, and path).
3. Under **Google Workspace Capabilities**, check the services to request — **Gmail (read-only)**, **Google Drive (read-only)**, **Google Calendar (read-only)**. Only check capabilities whose APIs you enabled in Step 1; each checkbox maps to the OAuth scope requested when a user connects their account.
4. Click **Save**. The connector status changes to **Configured** and users can connect their Google accounts.

> **The Redirect URI must match exactly.** The most common setup error is a mismatch between the Redirect URI entered here and the Authorized redirect URI in Google Cloud Console. If your users reach Kamiwaza at more than one hostname, add each one as an Authorized redirect URI in Google Cloud and set this field to the host they actually use.

> **Sensitive/restricted scopes.** Gmail and Drive read-only are *restricted* Google scopes. With an **Internal** consent screen (same Workspace org) this is fine; for an **External** app, Google may require app verification before users outside your test-user list can grant access.

Rotating credentials: To change the Client Secret (or Client ID), open **Configure**, enter the new value(s), and **Save** — leave a field blank to keep its stored value. The non-secret Redirect URI is always saved, so you can rebind the hostname without re-entering secrets.

> Need a connector that is not listed? Contact Kamiwaza Support to discuss roadmap status or professional-services extensions.

Best Practices

1. **Application Registration**: Create dedicated Azure AD applications (or equivalent) for Kamiwaza connectors rather than reusing existing applications
2. **Principle of Least Privilege**: Request only the minimum API permissions needed for the connector's functionality
3. **Credential Security**: Store application credentials securely and rotate them regularly
4. **User Communication**: Inform users about what data and permissions will be requested before they connect their accounts
5. **Testing**: Test connector configuration with a test account before making it available to all users
6. **Documentation**: Document the required Azure AD permissions and configuration steps for your team
7. **Monitoring**: Regularly review which users have connected accounts and ensure connectors remain properly configured

Troubleshooting

Connector Configuration Fails

- Verify you have administrative access to the Kamiwaza platform
- Check that all required fields are provided (Client ID, Client Secret, etc.)
- Ensure application credentials are valid and not expired
- Verify the Azure AD application has the required permissions configured
- Review error messages in the UI for specific validation failures

Users Cannot Connect Their Accounts

- Verify the connector shows as "Configured" in the admin interface
- Check that the OAuth redirect URIs are correctly configured in Azure AD (for Microsoft 365)
- Ensure the application credentials are still valid and have not expired
- Verify network connectivity between Kamiwaza and the external service

Authentication Errors

- Check that the Client ID and Client Secret are correct
- Verify the redirect URI in Azure AD matches Kamiwaza's callback URL
- Ensure the Azure AD application permissions are properly configured
- Check if the application requires admin consent for certain permissions

Google Workspace: a capability needs reconnect after enabling an API

If users connect successfully but a capability (Drive, Gmail, or Calendar) does not work — or the connection reports that it needs re-authentication with a message naming a Google API — the corresponding API is not enabled in the Google Cloud project:

1. In Google Cloud Console, open **APIs & Services** → **Library** and enable the named API (**Gmail API**, **Google Drive API**, or **Google Calendar API**).
2. Ask affected users to **reconnect** their account — re-enabling the API alone does not retroactively repair an existing connection.

Google Workspace: `redirect_uri_mismatch`

A `redirect_uri_mismatch` error during the Google consent flow means the connector's **Redirect URI** does not exactly match an **Authorized redirect URI** on the Google Cloud OAuth client. Confirm both are identical — scheme, host, and the `/api/connectors/public/google/callback` path — and that you edited the correct OAuth client.

Related Documentation

- Distributed Data Engine (see online documentation) - Overview of the DDE ingestion framework
- Data Catalog (see online documentation) - Managing ingested datasets
- Security (see online documentation) - Security and access control documentation
- Observability (see online documentation) - Monitoring and logging

Data Catalog and Secrets

Kamiwaza provides a centralized catalog for datasets, containers, and secrets. The catalog stores metadata, supports DataHub URNs, and is protected by access controls so only authorized users can view or change assets.

In current deployments, the catalog is the standard place to reference shared data assets and secret metadata across models, connectors, applications, and tools.

Core entities

Datasets

Datasets are the primary catalog entity. They are identified by DataHub URNs and can include schema metadata.

Containers

Containers are logical groupings of datasets (for example, a project or domain). They also use DataHub URNs and support dataset membership APIs. They also map well to things like S3 buckets, file folders, etc.

Secrets

Secrets are stored as catalog entities with encrypted values. Secret *metadata* is accessible via the API, but secret *values are never returned* through HTTP endpoints.

The current catalog backend encrypts secret values with a single Fernet key. It does not provide DEK/KEK envelope encryption or encryption-key rotation. Replacing a stored secret's value does not rotate that encryption key.

URN handling (DataHub)

Some DataHub URNs include forward slashes, which can conflict with path routing. Kamiwaza supports three approaches:

1. **V2 regex path endpoints (recommended)**
2. **Query-parameter endpoints (/by-urn)**
3. **Legacy path endpoints (simple URNs only)**

Dataset endpoints

```
POST   /api/catalog/datasets/

GET    /api/catalog/datasets/v2/{dataset_urn}
GET    /api/catalog/datasets/v2/{dataset_urn}/schema
PATCH  /api/catalog/datasets/v2/{dataset_urn}
DELETE /api/catalog/datasets/v2/{dataset_urn}
PUT    /api/catalog/datasets/v2/{dataset_urn}/schema

GET    /api/catalog/datasets/by-urn?urn={full_urn}
GET    /api/catalog/datasets/by-urn/schema?urn={full_urn}
PATCH  /api/catalog/datasets/by-urn?urn={full_urn}
DELETE /api/catalog/datasets/by-urn?urn={full_urn}
PUT    /api/catalog/datasets/by-urn/schema?urn={full_urn}

GET    /api/catalog/datasets/{encoded_urn}
PATCH  /api/catalog/datasets/{encoded_urn}
DELETE /api/catalog/datasets/{encoded_urn}
```

Container endpoints

```
GET    /api/catalog/containers/v2/{container_urn}
PATCH  /api/catalog/containers/v2/{container_urn}
DELETE /api/catalog/containers/v2/{container_urn}
POST   /api/catalog/containers/v2/{container_urn}/datasets
DELETE /api/catalog/containers/v2/{container_urn}/datasets/{dataset_urn}

GET    /api/catalog/containers/by-urn?urn={full_urn}
PATCH  /api/catalog/containers/by-urn?urn={full_urn}
DELETE /api/catalog/containers/by-urn?urn={full_urn}
```

Secret endpoints

```
POST   /api/catalog/secrets
GET    /api/catalog/secrets
GET    /api/catalog/secrets/v2/{secret_urn}
DELETE /api/catalog/secrets/v2/{secret_urn}

GET    /api/catalog/secrets/by-urn?urn={full_urn}
DELETE /api/catalog/secrets/by-urn?urn={full_urn}
```

Note: Secret values are never exposed in API responses. Use secrets in connectors and deployments by reference.

Register a dataset

This example registers one CSV object in S3, or an S3-compatible store, so the Retrieval Service (see online documentation) can read it. The credential is stored as a catalog secret and referenced by URN; it never appears in the dataset.

The examples use `API=https://kamiwaza.example.com/api` and an access token in `TOKEN`.

1. Store the credential. The secret's owner is your user URN:

```
OWNER=$(curl --fail --silent --show-error -H "Authorization: Bearer $TOKEN" \
  "$API/auth/users/me" | jq -er '"urn:li:corpuser:" + .sub')

VALUE=$(jq -nc --arg id "$AWS_ACCESS_KEY_ID" --arg key "$AWS_SECRET_ACCESS_KEY" \
  '{aws_access_key_id: $id, aws_secret_access_key: $key}')

SECRET_URN=$(jq -n --arg owner "$OWNER" --arg value "$VALUE" \
    '{name: "sales-s3-reader", owner: $owner, value: $value}' \
  | curl --fail --silent --show-error -X POST "$API/catalog/secrets/" \
      -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
      --data-binary @- | jq -er .)
```

Use the URN the call returns. A secret with the same name returns `409`. A wrong owner is not rejected here; it surfaces in step 2 as `403 dataset_secret_reference_denied`, because only the owner can use the secret.

2. Register the dataset. `path` names one object. Set `region` for AWS; for an S3-compatible store, also set `endpoint` to its URL, including scheme and port:

```
jq -n --arg secret "$SECRET_URN" '{
  name: "my-bucket/data/sales.csv",
  platform: "s3",
  properties: {
    path: "s3://my-bucket/data/sales.csv",
    format: "csv",
    region: "us-east-1",
    # endpoint: "https://s3.example.com:9000",   # S3-compatible stores only
    credential_secret: $secret
  }
}' | curl --fail --silent --show-error -X POST "$API/catalog/datasets/" \
      -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
      --data-binary @- | jq -r .
# urn:li:dataset:(urn:li:dataPlatform:s3,my-bucket/data/sales.csv,PROD)
```

`format` is `csv`, `parquet`, `json`, `text`, or `binary`. The response is the dataset URN, in the form `urn:li:dataset:(urn:li:dataPlatform:<platform>,<name>,<environment>)`; `environment` defaults to `PROD`. You become the dataset's owner.

3. Check it. A local retrieval job returns the object's rows:

```
curl --fail --silent --show-error -X POST "$API/retrieval/jobs" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"dataset_urn": "urn:li:dataset:(urn:li:dataPlatform:s3,my-
bucket/data/sales.csv,PROD)", "transport": "inline", "limit_rows": 10}' | jq .
```

Kamiwaza reaches the store from inside the cluster, so pods must resolve its hostname (see DNS (see online documentation)) and the cluster must allow egress to it. If the store's certificate is not issued by a publicly trusted CA, publish its CA in `kamiwaza-edge-trust` as for the edge certificate (see online documentation).

Access control

- **Datasets**: Protected by relationship-based access control (ReBAC). Owners/editors/viewers can access dataset metadata and schema.
- **Containers**: Container creation and membership changes require admin permissions.
- **Secrets**: Protected by ReBAC; only authorized users can view or delete secret metadata.

Best practices

- Prefer `/v2/{urn}` or `/by-urn` endpoints for any URN that might contain `/` characters.
- Use secrets for credentials instead of embedding them in connector definitions, templates, or external endpoint payloads.
- Align dataset URNs with your DataHub naming conventions so discovery, lineage, and governance are consistent across tools.

Retrieval Service

The retrieval service exposes a job-based API for accessing dataset content. It supports inline responses for small datasets, server-sent events (SSE) for streaming, and gRPC for high-throughput consumers.

Supported sources

Retrieval adapters cover these data sources:

- Filesystem
- Amazon S3
- Hive
- Postgres
- Kafka
- Slack

Core workflow

1. **Create a retrieval job**
2. **Poll job status** (optional for streaming or gRPC)
3. **Stream results** (SSE) or connect via gRPC

Create a job

`POST /api/retrieval/jobs`

Key fields:

- `dataset_urn` (required)
- `transport` (auto, inline, sse, grpc)
- `columns`, `limit_rows` (optional)
- `options` for source-specific settings (for example, Kafka bootstrap servers)

Get job status

`GET /api/retrieval/jobs/{job_id}`

Returns status, progress, and dataset metadata. If the job uses gRPC, the response includes a short-lived token and endpoint.

Stream results (SSE)

`GET /api/retrieval/jobs/{job_id}/stream`

Streams data as `text/event-stream` chunks. Use this for larger datasets or when you need progressive delivery.

gRPC transport

If a job response includes a `grpc` block, use the token and endpoint with the retrieval gRPC service:

- Service: `kamiwaza.retrieval.v1.RetrievalService`
- Method: `StreamData`

The gRPC stream returns data chunks with metadata and a terminal marker.

Access control

Retrieval requests are authorized against the dataset URN. Users must have viewer (or higher) access to the dataset to create or view jobs.

Notes

- Use `transport=auto` to let the service choose between inline and streaming.
- For large datasets, prefer `sse` or `grpc` to avoid response size limits.

Observability Guide

Kamiwaza provides several ways to inspect deployment health, runtime errors, and security-relevant activity. In current customer deployments, the primary observability paths are the UI log viewer, Kubernetes-native logs, and optional OpenTelemetry forwarding into your existing monitoring platform.

Overview

Kamiwaza observability is built around three layers:

- **Deployment logs** surfaced through the Kamiwaza UI for models, apps, and tools
- **Platform logs** available through Kubernetes and your cluster logging workflow
- **OpenTelemetry (OTEL)** for exporting telemetry to external systems

Current public guidance assumes a Kubernetes deployment and does not rely on host-local log directories or bundled Grafana/Loki services.

Start Here

For most troubleshooting:

1. use the Kamiwaza UI log viewer first
2. check Kubernetes pod health and container logs if the issue is platform-wide
3. review any external OTEL destination if your environment forwards logs or traces out of cluster

UI Log Viewer

Kamiwaza includes a logger service that powers the UI log viewer and deployment log APIs.

Common endpoints behind the standard API gateway include:

- `GET /api/logger/deployments/all` - list deployment logs
- `GET /api/logger/deployments/type/{type}` - filter by deployment type
- `GET /api/logger/deployments/{type}/{id}` - fetch log content
- `GET /api/logger/deployments/{type}/{id}/patterns` - view detected error patterns
- `DELETE /api/logger/deployments/{type}/{id}?confirm=true` - delete a deployment log file
- `DELETE /api/logger/cleanup?days_old=30&dry_run=true` - preview or remove older logs

Use the log viewer in the UI when you want to:

- inspect model startup failures
- review app or tool launch problems

- check for common error signatures such as out-of-memory or runtime crashes

Kubernetes-Level Troubleshooting

For cluster administrators, platform issues are usually easiest to diagnose through Kubernetes.

Typical checks include:

- pod status in the Kamiwaza namespace
- container restart counts
- recent pod logs for the affected service
- ingress or routing behavior for the public domain

Use your environment's standard Kubernetes and platform tooling to inspect those signals. If your organization centralizes Kubernetes logs in another system, treat that system as the source of truth for cluster-level troubleshooting.

OpenTelemetry Integration

Kamiwaza supports OpenTelemetry-based export so customer environments can forward telemetry to existing monitoring systems.

Typical uses include:

- forwarding logs and traces to a central observability platform
- correlating platform events with the rest of your cluster telemetry
- retaining security and troubleshooting data outside the application UI

OTEL is controlled through deployment configuration rather than manual edits inside running containers.

Common settings include:

- `KAMIWAZA_OTEL_ENABLED`
- `CUSTOMER_OTLP_ENDPOINT`
- `CUSTOMER_OTLP_AUTH`
- `OTEL_EXPORTER_INSECURE`

Best practice:

- enable OTEL through deployment values or ConfigMaps
- keep OTLP credentials in Kubernetes Secrets or your approved secret manager
- use TLS-enabled collectors for production environments whenever possible

External Observability Platforms

Kamiwaza can be integrated with OTLP-compatible platforms such as Splunk, Datadog, Dynatrace, or another OpenTelemetry pipeline managed by your organization.

When using an external destination, validate:

- the endpoint is reachable from the cluster
- credentials are present and current
- exported telemetry appears in the expected tenant, index, or project

Log Types

The platform emits several categories of logs:

Application logs

Used for API operations, business logic, and service errors.

Typical examples:

- model deployment failures
- connector execution errors
- service exceptions

Audit and security logs

Used for security-sensitive operations and administrative actions.

Typical examples:

- authentication events
- access denials
- privileged operations

Deployment logs

Used for model, app, and tool runtime troubleshooting.

Typical examples:

- engine startup output
- container stderr or stdout
- launch-time readiness failures

Ingress and access logs

Used for request-level routing and gateway debugging.

Typical examples:

- upstream routing errors
- unexpected status codes
- repeated auth or callback failures

Troubleshooting Common Issues

A model, app, or tool fails to start

- start with the UI log viewer for that deployment
- look for detected patterns such as out-of-memory, startup timeout, or container exit
- if the problem affects multiple deployments, check Kubernetes pod status and cluster logs

A service is reachable but behaving incorrectly

- inspect application logs for the affected service
- compare the failing request path against the expected routed URL
- verify the deployment still matches the intended configuration

External OTEL destination is missing data

- confirm OTEL is enabled in the deployment
- verify the configured OTLP endpoint and auth settings
- inspect collector or exporter logs in the cluster
- confirm the destination platform is receiving data for the correct environment

Authentication or access issues need correlation

- review security and audit logs
- confirm the request path, user context, and deny behavior
- use the security validation docs when the issue is tied to ReBAC or identity setup

Configuration Reference

Common observability-related settings include:

Variable	Purpose
`KAMIWAZA_OTEL_ENABLED`	Enables OTEL export paths
`CUSTOMER_OTLP_ENDPOINT`	External OTLP destination
`CUSTOMER_OTLP_AUTH`	Auth header or token for the external OTLP destination
`OTEL_EXPORTER_INSECURE`	Allows insecure OTLP transport for narrowly scoped non-production cases
`KAMIWAZA_DISABLE_SYSLOG`	Controls syslog forwarding where that compatibility mode is still in use

Manage these through deployment values, ConfigMaps, and Secrets, not through ad hoc edits on running containers.

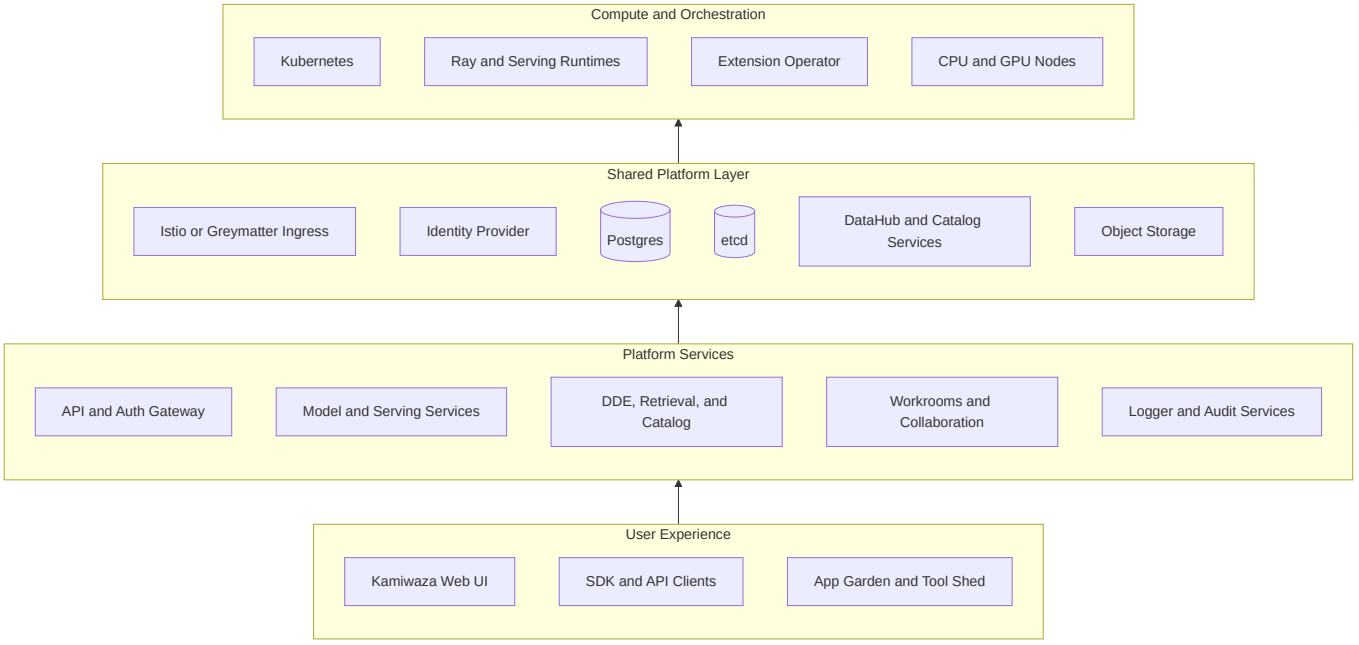
Related Docs

- Quickstart (see online documentation)
- Configuration Reference (see online documentation)
- Administrator Guide (see online documentation)

Platform Architecture Overview

Kamiwaza is a Kubernetes-based AI platform that combines model serving, governed data access, application deployment, and security controls behind a single customer-facing domain. This page provides a high-level view of the current architecture used in supported deployments.

System Architecture Diagram



Architecture Layers

Experience layer

This is how users and client applications interact with Kamiwaza.

- **Kamiwaza Web UI** for administration, models, workrooms, apps, tools, and logs
- **SDK and API clients** for programmatic access
- **App Garden and Tool Shed** for launching applications and tool servers behind managed routes

Services layer

This layer contains the platform APIs and business logic.

- **API and Auth Gateway** for authenticated access, routing, and policy enforcement
- **Model and Serving Services** for model lifecycle and inference
- **DDE, Retrieval, and Catalog** for ingestion, discovery, secret references, and retrieval flows
- **Workrooms and Collaboration** for shared workspace functionality

- **Logger and Audit Services** for runtime troubleshooting and security evidence

Shared platform layer

This layer provides the shared services the platform depends on.

- **Istio or Greymatter** for ingress and routing, depending on environment
- **Identity provider integration** for authentication
- **Postgres** as the persistent database
- **etcd** for cluster coordination and runtime configuration
- **DataHub and related catalog services** for metadata-backed catalog workflows
- **Object storage** for uploaded files, workroom context, and related assets

Compute and orchestration layer

This layer runs the workloads that power the platform.

- **Kubernetes** as the primary deployment target
- **Ray and serving runtimes** for model execution and distributed work
- **Extension Operator** for managed application and tool deployment workflows
- **CPU and GPU nodes** that host inference and platform workloads

Technology Stack

Category	Technologies
Backend	Python, FastAPI, Ray
Frontend	React
Data and metadata	Postgres, etcd, DataHub
Routing and ingress	Istio, or Greymatter on supported external-mesh deployments
Deployment	Kubernetes, Helm

Design Principles

- **Single customer-facing domain** with path-based runtime routing as the standard deployment model
- **Security by configuration** through identity integration, access controls, consent, and auditability
- **Composable services** so model serving, retrieval, catalog, apps, and tools can evolve independently
- **Operational visibility** through deployment logs, cluster logs, and OTEL-compatible export paths

Core Components & Concepts

The Kamiwaza platform is composed of several key components and conceptual systems that work together to provide model serving, governed data access, and managed application runtimes. This page describes the most important of those building blocks in the current platform.

Model & Data Handling

Vector Databases

- **What it is:** A specialized database for storing and retrieving high-dimensional vector data, such as embeddings generated from text or images. Kamiwaza can integrate with vector backends such as **Milvus** and **Qdrant**.
- **Why it matters:** Vector databases are the engine behind similarity search, Retrieval-Augmented Generation (RAG), and other semantic retrieval workflows. Kamiwaza's abstraction layer lets deployments adopt supported backends without changing user-facing workflows.

Embeddings Management

- **What it is:** The process of generating, storing, and managing vector embeddings for your data. Kamiwaza provides platform services that automate embedding generation using a dedicated, standalone OpenAI-compatible `llama.cpp` embedding service deployed via Helm (replacing legacy in-process fallbacks).
- **Why it matters:** Consistent and efficient embedding management is fundamental to vector search quality and retrieval performance. A dedicated embedding service reduces container footprint and improves service decoupling compared to in-process fallbacks.

Data Catalog

- **What it is:** A centralized inventory of data assets, containers, and secret metadata. Kamiwaza integrates with **DataHub**-backed catalog workflows and uses URN-based references across the platform.
- **Why it matters:** As AI systems grow, so does the data they consume. A data catalog improves discoverability, governance, and access control across models, retrieval flows, connectors, applications, and tools.

Orchestration & Serving

Orchestration Engine

- **What it is:** The control and execution layer that manages model launches, runtime coordination, and distributed work. Kamiwaza uses **Ray** and related serving runtimes to handle that workload.
- **Why it matters:** This layer keeps inference, background work, and runtime routing coordinated across cluster resources.

Model Serving

- **What it is:** The process of taking a model and making it available for real-time inference via a managed API route. Kamiwaza supports multiple serving engines such as **vLLM**, **llama.cpp**, and **MLX**, depending on environment and model type.
- **Why it matters:** Different models have different hardware and runtime requirements. Multi-engine support helps the platform balance compatibility, performance, and cost.

API Gateway

- **What it is:** A single, unified entry point for platform APIs. Kamiwaza uses **FastAPI** plus ingress routing to expose those services through a consistent customer-facing interface.
- **Why it matters:** A gateway simplifies development by providing one access surface for models, retrieval, apps, tools, logging, and administration. It is also the natural place to enforce authentication, rate limiting, and logging.

Security & Operations

Identity & Access Management

- **What it is:** The system that handles user authentication (who you are) and authorization (what you are allowed to do).
- **Why it matters:** Robust security is non-negotiable in an enterprise setting. Kamiwaza's IAM services ensure that only authorized users and applications can access sensitive data and models.

Caching

- **What it is:** A high-speed storage layer used for session state, temporary runtime coordination, and frequently accessed data.
- **Why it matters:** Caching dramatically improves the performance and responsiveness of AI applications, especially those with high request volumes, leading to a better user experience and lower operational costs.

Core Services

Kamiwaza's backend is built as a collection of specialized services, each handling a specific aspect of platform behavior. Together they support model serving, governed data access, application deployment, workrooms, and security controls.

Service Architecture

The backend follows a consistent pattern where each service is self-contained and follows the structure:

```
service/
├── api.py      # FastAPI router
├── models/     # SQLAlchemy ORM
├── schemas/    # Pydantic DTOs
└── services.py # Business logic
```

This modular approach helps with:

- **Separation of concerns** - Each service has a clear, focused responsibility
- **Scalability** - Services can be scaled independently based on demand
- **Maintainability** - Changes to one service do not affect others
- **Testability** - Each service can be tested in isolation

Core Services Overview

🤖 Models Service

Manages the lifecycle of AI models including registration, download, deployment, and serving. It coordinates runtime selection and exposes model APIs through the platform routing layer.

📄 Catalog Service

Provides metadata-backed management for datasets, containers, and secrets. It enables shared references across models, connectors, apps, tools, and retrieval workflows.

📥 Ingestion and DDE Services

Handle connector configuration, scheduled ingestion, and document indexing flows. These services are the bridge between external data sources and retrieval-ready content inside the platform.

🔎 Retrieval Service

Provides job-based access to dataset content for downstream search, analysis, and RAG workflows. It supports inline, streaming, and gRPC-style retrieval patterns where available.

🧠 Embedding and Vector Services

Handle embedding generation and vector-backed retrieval infrastructure used by semantic search and retrieval workflows.

🔐 Authentication Service

Manages authenticated sessions, identity-provider integration, and access control enforcement for platform APIs.

🧱 Workroom Services

Support collaborative workspaces, presence information, shared context, and workroom-specific access behavior.

📈 Logger and Audit Services

Provide deployment logs, operational events, and audit evidence used for troubleshooting and security review.

🌱 Garden Services

Power App Garden and Tool Shed deployment workflows, including template resolution, managed runtime configuration, and routed access to launched workloads.

Service Communication

All services communicate through:

- **FastAPI routers** for HTTP API endpoints
- **Ray and serving runtimes** for distributed model execution
- **Shared platform stores** such as Postgres and etcd
- **Ingress and routing layers** for user-facing access to models, apps, and tools

Integration Patterns

Services are designed to work together seamlessly:

- **Models + Gardens** - Expose deployed models to applications and tools through managed template variables
- **Ingestion + Catalog + Retrieval** - Bring external data into the platform and make it queryable
- **Catalog + Secrets + External Endpoints** - Reuse governed secret references across integrations
- **Auth + ReBAC + Service APIs** - Enforce the correct user and tenant boundaries across platform workflows

- **Logger + All Services** - Monitor and troubleshoot platform interactions

Next Steps

To learn more about working with these services:

- Explore the Models (see online documentation) and Distributed Data Engine (see online documentation) documentation
- Build a complete RAG pipeline (see online documentation) using multiple services
- Review the Platform Overview (see online documentation) for architectural context
- Check out Use Cases (see online documentation) for practical implementation examples

Use Cases

Kamiwaza enables a wide variety of AI applications and workflows. This section provides practical guidance for implementing common use cases, complete with step-by-step instructions, best practices, and example code.

What You Will Find Here

Each use case guide includes:

- **Overview** - What the use case accomplishes and when to use it
- **Prerequisites** - Required models, services, and setup steps
- **Implementation** - Step-by-step instructions with code examples
- **Best Practices** - Tips for optimization and production deployment
- **Troubleshooting** - Common issues and solutions

Featured Use Cases

📖 Building a RAG Pipeline (see online documentation)

Learn how to create a Retrieval-Augmented Generation (RAG) system that combines your documents with large language models to provide accurate, context-aware responses.

What you will build:

- Document ingestion and preprocessing
- Vector embeddings for semantic search
- Retrieval system for relevant context
- LLM integration for response generation

Perfect for: Customer support, internal knowledge bases, document Q&A systems

Coming Soon

Additional use case guides are in development:

🤖 Multi-Agent Systems

Build sophisticated AI agents that can collaborate to solve complex tasks, with coordination, memory, and tool usage capabilities.

🎯 Custom Model Fine-tuning

Deploy and serve your own fine-tuned models, including setup for training workflows and model versioning.

🔄 Real-time Data Processing

Stream processing pipelines that combine AI models with live data feeds for real-time insights and actions.

📊 Analytics and Monitoring

Comprehensive monitoring setups for AI applications, including performance tracking, model drift detection, and usage analytics.

🌐 Multi-modal Applications

Applications that work with text, images, and other data types using Kamiwaza's flexible model serving capabilities.

🔗 API Integration Patterns

Common patterns for integrating Kamiwaza with existing systems, including webhooks, batch processing, and microservice architectures.

Getting Started

1. **Review the Prerequisites** - Make sure you have Kamiwaza installed and running
2. **Choose Your Use Case** - Pick the guide that matches your needs
3. **Follow Along** - Each guide includes working code and examples
4. **Adapt and Extend** - Use the patterns as a foundation for your specific requirements

Need Help?

If you are looking for a specific use case that is not covered yet:

- Join the Discord community (https://discord.gg/cVGBS5rD2U) to ask questions
- Explore the SDK documentation (see online documentation) for programmatic approaches
- Check out the Models (see online documentation) and App Garden (see online documentation) sections for additional patterns

---

Have a use case you would like to see documented? Ask on the Discord community (https://discord.gg/cVGBS5rD2U) or contact Kamiwaza (https://kamiwaza.ai/contact).

Building a RAG Pipeline

Retrieval-Augmented Generation (RAG) combines the power of large language models with your own documents and data to provide accurate, contextual responses. This guide walks you through building a complete RAG pipeline using Kamiwaza's core services.

What You Will Build

By the end of this guide, you will have:

- **Document ingestion system** that processes various file formats
- **Embedding pipeline** that converts text to vector representations
- **Vector search system** for finding relevant context
- **LLM integration** that generates responses using retrieved context
- **Web interface** for querying your documents

Prerequisites

Before starting, ensure you have:

- Kamiwaza installed and running (Installation Guide (see online documentation))
- At least 16GB of available RAM
- Sample documents (markdown format) to process
- Basic familiarity with Python (for SDK examples)

Architecture Overview

A RAG pipeline consists of four main components:

Step 1: Deploy Required Models

Deploy an embedding model for vectorizing text and a language model for generating responses.

Deploy an Embedding Model

The embedding model will be automatically loaded when you create an embedder - no manual deployment needed:

```python
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient(base_url="https://your-kamiwaza.example/api")

# The embedding model will be automatically loaded when you create an embedder
# This happens seamlessly in the background
embedder = client.embedding.get_embedder(
    model="BAAI/bge-base-en-v1.5",
    provider_type="huggingface_embedding"
)

print("✅ Embedding model ready for use")
```

Deploy a Language Model

Deploy a language model using Kamiwaza for response generation:

```python
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient(base_url="https://your-kamiwaza.example/api")

# Search for a suitable language model
model_repo = "Qwen/Qwen3-0.6B-GGUF"  # Small efficient model
models = client.models.search_models(model_repo, exact=True)
print(f"Found model: {models[0]}")

# Download the model (this may take a few minutes)
print("Downloading model...")
client.models.initiate_model_download(model_repo)
client.models.wait_for_download(model_repo)
print("✅ Model download complete")

# Deploy the model
print("Deploying model...")
deployment_id = client.serving.deploy_model(repo_id=model_repo)
print(f"✅ Model deployed with ID: {deployment_id}")

# Get OpenAI-compatible client for the deployed model
openai_client = client.openai.get_client(repo_id=model_repo)
print("✅ OpenAI-compatible client ready")
```

Check Deployment Status

```python
# List active deployments to verify
deployments = client.serving.list_active_deployments()
for deployment in deployments:
    print(f"✅ {deployment.m_name} is {deployment.status}")
    print(f"   Endpoint: {deployment.endpoint}")
```

Step 2: Document Ingestion Pipeline

Create a pipeline to process documents, chunk them, and generate embeddings.

Document Processing Script

```
import os
from pathlib import Path
from typing import List, Dict
from kamiwaza_sdk import KamiwazaClient

class RAGPipeline:
    def __init__(self, base_url="https://your-kamiwaza.example/api"):
        self.client = KamiwazaClient(base_url=base_url)
        self.embedding_model = "BAAI/bge-base-en-v1.5"  # Use a proven working model
        self.collection_name = "documents"

        # Initialize global embedder to prevent cleanup between operations
        self.embedder = self.client.embedding.get_embedder(
            model=self.embedding_model,
            provider_type="huggingface_embedding"
        )
        print(f"✅ RAG Pipeline initialized with model: {self.embedding_model}")

    def add_documents_to_catalog(self, filepaths: List[str]) -> List:
        """Add documents to the Kamiwaza catalog."""
        datasets = []

        for filepath in filepaths:
            try:
                # Create dataset for each file
                dataset = self.client.catalog.create_dataset(
                    dataset_name=filepath,
                    platform="file",
                    environment="PROD",
                    description=f"RAG document: {Path(filepath).name}"
                )

                if dataset.urn:
                    datasets.append(dataset)
                    print(f"✅ Added to catalog: {Path(filepath).name}")

            except Exception as e:
                print(f"❌ Error adding {filepath}: {str(e)}")

        return datasets

    def process_document(self, file_path: str):
        """Process a single document: read, chunk, embed, and store."""
        doc_path = Path(file_path)

        if not doc_path.exists():
            raise FileNotFoundError(f"File not found: {doc_path}")

        # Read document content
        with open(doc_path, 'r', encoding='utf-8') as f:
            content = f.read()

        print(f"📄 Processing document: {doc_path.name}")
        print(f"   - Size: {len(content)} characters")

        # Chunk the document using SDK
        chunks = self.embedder.chunk_text(
            text=content,
```

```
            max_length=1024,  # Token-based chunking
            overlap=102       # 10% overlap
        )
        print(f"   - Created {len(chunks)} chunks")

        # Generate embeddings for all chunks
        embeddings = self.embedder.embed_chunks(chunks)
        print(f"   - Generated {len(embeddings)} embeddings")

        # Prepare metadata for each chunk
        metadata_list = []
        for i, chunk in enumerate(chunks):
            # Truncate chunk text if needed to fit storage limits
            chunk_text = chunk[:900] + "..." if len(chunk) > 900 else chunk

            metadata = {
                # Required autofields
                "model_name": self.embedding_model,
                "source": str(doc_path),
                "offset": i,
                "filename": doc_path.name,

                # Custom fields for better search
                "chunk_text": chunk_text,  # Store the actual text
                "chunk_index": i,
                "chunk_size": len(chunk),
                "document_title": doc_path.stem
            }
            metadata_list.append(metadata)

        # Define custom fields for the collection schema
        field_list = [
            ("chunk_text", "str"),
            ("chunk_index", "int"),
            ("chunk_size", "int"),
            ("document_title", "str")
        ]

        # Insert vectors using SDK
        self.client.vectordb.insert(
            vectors=embeddings,
            metadata=metadata_list,
            collection_name=self.collection_name,
            field_list=field_list
        )

        print(f"✅ Successfully stored {len(chunks)} chunks in collection
'{self.collection_name}'")
        return len(chunks)

# Usage
pipeline = RAGPipeline()

# Example: Process documents
DOCUMENT_PATHS = [
    "./docs/intro.md",
    "./docs/models/overview.md",
    "./docs/architecture/overview.md",
    "./docs/architecture/architecture.md",
    "./docs/architecture/components.md"
    # Add more documents as needed
]
```

```
# Optional: Add to catalog first
datasets = pipeline.add_documents_to_catalog(DOCUMENT_PATHS)

# Process each document
total_chunks = 0
for doc_path in DOCUMENT_PATHS:
    try:
        chunks = pipeline.process_document(doc_path)
        total_chunks += chunks
    except Exception as e:
        print(f"❌ Error processing {doc_path}: {str(e)}")

print(f"\n🎉 Total chunks processed: {total_chunks}")
```

Step 3: Implement Retrieval and Generation

Create the query interface that retrieves relevant documents and generates responses.

```
from typing import List, Dict
from kamiwaza_sdk import KamiwazaClient

class RAGQuery:
    def __init__(self, base_url="https://your-kamiwaza.example/api",
chat_model_repo="Qwen/Qwen3-0.6B-GGUF"):
        self.client = KamiwazaClient(base_url=base_url)
        self.embedding_model = "BAAI/bge-base-en-v1.5"
        self.chat_model_repo = chat_model_repo
        self.collection_name = "documents"

        # Initialize embedder for query processing
        self.embedder = self.client.embedding.get_embedder(
            model=self.embedding_model,
            provider_type="huggingface_embedding"
        )

        # Get OpenAI-compatible client for the deployed chat model
        try:
            self.openai_client =
self.client.openai.get_client(repo_id=self.chat_model_repo)
            print(f"✅ RAG Query system initialized with chat model:
{self.chat_model_repo}")
        except Exception as e:
            print(f"⚠️ Warning: Could not initialize chat model client: {e}")
            print(f"   Make sure the model {self.chat_model_repo} is deployed")
            self.openai_client = None

    def semantic_search(self, query: str, limit: int = 5) -> List[Dict]:
        """Perform semantic search on the document collection."""
        print(f"🔍 Searching for: '{query}'")
        print(f"   - Collection: {self.collection_name}")
        print(f"   - Max results: {limit}")

        # Generate embedding for the query
        query_embedding = self.embedder.create_embedding(query).embedding

        # Perform vector search using SDK
        results = self.client.vectordb.search(
            query_vector=query_embedding,
            collection_name=self.collection_name,
            limit=limit,
            output_fields=[
                "source", "offset", "filename", "model_name",
                "chunk_text", "chunk_index", "chunk_size", "document_title"
            ]
        )

        print(f"✅ Found {len(results)} relevant chunks")
        return results

    def format_context(self, search_results: List[Dict]) -> str:
        """Format search results into context for LLM."""
        context_parts = []

        for result in search_results:
            # Extract metadata
            if hasattr(result, 'metadata'):
                metadata = result.metadata
            elif isinstance(result, dict) and 'metadata' in result:
```

```
                metadata = result['metadata']
            else:
                metadata = {}

            # Get chunk text and source info
            chunk_text = metadata.get('chunk_text', '')
            filename = metadata.get('filename', 'Unknown')
            document_title = metadata.get('document_title', filename)

            if chunk_text:
                context_parts.append(f"Document: {document_title}\n{chunk_text}")

        return "\n\n".join(context_parts)

    def generate_response(self, query: str, context: str) -> str:
        """Generate response using retrieved context and deployed Kamiwaza model."""
        if not self.openai_client:
            return f"[Error: Chat model not available. Please deploy
{self.chat_model_repo} first]"

        prompt = f"""Based on the following context, answer the user's question. If the
context doesn't contain enough information to answer the question, say so.

        Context:
        {context}

        Question: {query}

        Answer:"""

        try:
            # Use the deployed Kamiwaza model via OpenAI-compatible interface
            response = self.openai_client.chat.completions.create(
                messages=[
                    {"role": "user", "content": prompt}
                ],
                model="model",  # Use "model" as the model name for Kamiwaza OpenAI
interface
                max_tokens=500,
                temperature=0.7,
                stream=False
            )

            return response.choices[0].message.content

        except Exception as e:
            return f"[Error generating response: {str(e)}]"

    def query(self, user_question: str, limit: int = 5) -> Dict:
        """Complete RAG query pipeline."""
        print(f"🤖 Processing RAG query: {user_question}")

        # Search for relevant documents
        search_results = self.semantic_search(user_question, limit=limit)

        # Format context for LLM
        context = self.format_context(search_results)

        # Generate response (you'll need to implement LLM integration)
        response = self.generate_response(user_question, context)

        # Prepare sources information
```

```
        sources = []
        for result in search_results:
            metadata = result.metadata if hasattr(result, 'metadata') else
result.get('metadata', {})
            score = result.score if hasattr(result, 'score') else result.get('score',
0.0)

            sources.append({
                'filename': metadata.get('filename', 'Unknown'),
                'document_title': metadata.get('document_title', ''),
                'chunk_index': metadata.get('chunk_index', 0),
                'score': score
            })

        return {
            'question': user_question,
            'answer': response,
            'context': context,
            'sources': sources,
            'num_results': len(search_results)
        }
```

Step 4: Example Queries

Test the RAG system with an example query to demonstrate its capabilities:

```
# Example query to test your RAG system
rag = RAGQuery()

query_response = rag.query("What is one cool thing about Kamiwaza?")

print(query_response['answer'])
```

Step 5: Production Considerations

When moving your RAG system to production, consider these key aspects:

Resource Management

```
# Monitor system resources and manage deployments
def monitor_system_health():
    """Monitor system health and resource usage."""
    client = KamiwazaClient(base_url="https://your-kamiwaza.example/api")

    # Check active deployments
    deployments = client.serving.list_active_deployments()
    print(f"📊 Active Deployments: {len(deployments)}")

    for deployment in deployments:
        print(f"   - {deployment.m_name}: {deployment.status}")
        print(f"     Endpoint: {deployment.endpoint}")

    # Check vector collections
    collections = client.vectordb.list_collections()
    print(f"📚 Vector Collections: {collections}")

# Run health check
monitor_system_health()
```

Clean up

When done, stop the model deployment to free resources

```
def cleanup_rag_system(chat_model_repo="Qwen/Qwen3-0.6B-GGUF"):
    """Stop model deployments to free up resources."""
    client = KamiwazaClient(base_url="https://your-kamiwaza.example/api")

    try:
        success = client.serving.stop_deployment(repo_id=chat_model_repo)
        if success:
            print(f"✅ Stopped deployment for {chat_model_repo}")
        else:
            print(f"❌ Failed to stop deployment for {chat_model_repo}")
    except Exception as e:
        print(f"❌ Error stopping deployment: {e}")

cleanup_rag_system()
```

Best Practices

Document Processing

- **Chunk Size**: Keep chunks between 200-800 tokens for optimal retrieval
- **Overlap**: Add 50-100 token overlap between chunks to preserve context

- **Metadata**: Include rich metadata (source, date, author) for filtering
- **Preprocessing**: Clean text, remove headers/footers, handle special characters

Vector Search Optimization

- **Index Tuning**: Adjust index parameters based on collection size
- **Reranking**: Use a reranking model for better result quality
- **Hybrid Search**: Combine vector search with keyword matching
- **Filtering**: Use metadata filters to narrow search scope

LLM Integration

- **Context Window**: Stay within model's context limits
- **Prompt Engineering**: Design clear, specific system prompts
- **Temperature**: Use lower values (0.1-0.3) for factual responses
- **Citations**: Always include source attribution in responses

Troubleshooting

Common Issues

Poor Retrieval Quality

- Check embedding model performance on your domain
- Adjust chunk size and overlap
- Try different similarity metrics (cosine vs. dot product)
- Consider domain-specific fine-tuning

Slow Query Performance

- Optimize vector index parameters
- Reduce `top_k` in retrieval
- Use GPU acceleration for embeddings
- Implement result caching

Inaccurate Responses

- Improve prompt engineering
- Increase retrieved context size
- Use a larger/better language model
- Add response validation logic

Next Steps

Now that you have a working RAG pipeline with Kamiwaza-deployed models:

- **Try Different Models**: Experiment with larger models like `Qwen/Qwen3-32B-GGUF` for better response quality
- **Optimize Retrieval**: Experiment with different embedding models, chunk sizes, and similarity thresholds
- **Add Streaming**: Use `stream=True` in the chat completions for real-time response streaming
- **Implement Reranking**: Add semantic reranking for better result quality
- **Scale Your System**: Deploy multiple model instances using Kamiwaza's distributed architecture (see online documentation)
- **Monitor Performance**: Add logging and metrics to track query performance and model usage

Key Benefits of This SDK-Based Approach

- **Simplified Integration**: No need for manual HTTP requests - the SDK handles all API communication
- **Automatic Schema Management**: Collections and schemas are created automatically based on your data
- **Built-in Best Practices**: The SDK incorporates proven patterns for chunking, embedding, and vector storage
- **Catalog Integration**: Documents are managed through Kamiwaza's catalog system for better organization
- **Production Ready**: SDK handles error cases, retries, and connection management

Your RAG pipeline is now ready to answer questions using your own documents! The combination of Kamiwaza's SDK with proper document processing creates a robust foundation for production RAG applications.

Help & Fixes

This page provides resources for getting help with Kamiwaza and solutions to common issues you might encounter.

Getting Help

If you have questions or run into issues, use any of these channels:

- Join the Discord community (https://discord.gg/cVGBS5rD2U)
- Visit the Kamiwaza website (https://www.kamiwaza.ai/)
- Visit the GitHub organization (https://github.com/kamiwaza-ai)
- Try the client SDK (https://pypi.org/project/kamiwaza-sdk/)
- Contact the support team (https://portal.kamiwaza.ai/_hcms/mem/login?redirect_url=https%3A%2F%2Fportal.kamiwaza.ai%2Ftickets-view)

Kamiwaza is committed to making your experience as smooth as possible.

Reporting Issues

When reporting issues to the support team or community, please include:

- **Environment Details**: OS version, Kubernetes distribution and version, hardware specs, and the Kamiwaza release version
- **Error Messages**: Complete error text and stack traces
- **Steps to Reproduce**: Detailed steps that led to the issue
- **Logs**: Relevant Kubernetes pod logs, platform logs, and UI error details
- **Configuration**: Any custom configuration or settings

This information enables faster and more accurate solutions.

Common Issues and Fixes

Installation Issues

Docker GPU Error: Could Not Select Device Driver

Problem: NVIDIA Container Runtime not found or misconfigured.

Solution:

- Ensure NVIDIA drivers are properly installed

- Install NVIDIA Container Toolkit
- Verify Docker can access GPU devices

Port Already in Use

Problem: Kamiwaza fails to start because required ports are occupied.

Solution:

- Check what's running on ports 3000, 8000, 5432, 19530, 9090
- Stop conflicting services or change Kamiwaza's port configuration
- Use `lsof -i :PORT_NUMBER` to identify processes using specific ports

Insufficient System Resources

Problem: Installation fails due to low disk space, RAM, or CPU cores.

Solution:

- Ensure at least 16GB RAM available
- Verify CPU supports required virtualization features

Model Deployment Issues

Gated Model Downloads and Rate Limits

Problem: Downloads fail for gated models (Llama, Mistral, etc.) or you encounter Hugging Face rate limit errors.

Symptoms:

- "Access denied" or "401 Unauthorized" when downloading certain models
- "Rate limit exceeded" errors during model downloads
- Gated models appear in search but fail to download

Solution: Add your Hugging Face token to your Kamiwaza deployment configuration:

1. Get a Hugging Face token from huggingface.co/settings/tokens (https://huggingface.co/settings/tokens) (a **read** token is sufficient)
2. For gated models, accept the license terms on the model's Hugging Face page
3. Add the token to the Secret or deployment values used for your Kamiwaza environment:

   ```
   HF_TOKEN="hf_your_token_here"
   ```

4. Apply the updated configuration using your standard cluster release workflow and confirm the model-serving pods restart successfully.
5. If you need setup details for secrets or deployment values, refer to the Configuration Reference (see online documentation) and the Administrator Guide (see online documentation).

Model Deployment Failures

Problem: Models fail to deploy or become unavailable.

Solutions:

- **Model not found**: Ensure the model exists in your catalog or use Novice Mode
- **Checkpoint too large for VRAM**: Choose a smaller/quantized variant (AWQ, MLX, GGUF) or reduce batch size
- **Service unavailable/port errors**: Stop/Remove and redeploy the model
- **Outdated catalog**: Refresh the Models page or restart the server

Performance Problems

Problem: Slow responses or high resource usage.

Solutions:

- **Slow responses**: Use faster models or quantized variants; reduce max tokens and context length
- **High memory/OOM**: Lower batch size, context length, and KV cache; use lower-VRAM variants
- **Cold starts**: First request may be slower; send a short warm-up prompt after deploy

SDK and API Issues

Module Import Error

Problem: `ModuleNotFoundError: No module named 'kamiwaza_sdk'` when using notebooks and Kamiwaza SDK.

Solution:

```
!pip uninstall -y kamiwaza kamiwaza-sdk
!pip install kamiwaza-sdk
```

Then restart the notebook kernel or Python session before importing `kamiwaza_sdk` again.

App Garden Issues

App Not Showing Latest Version

Problem: An app in App Garden is not displaying the latest content or updates due to cache TTL.

Solution: Force a cache refresh by syncing remote templates:

```
# Step 1: Get an auth token
TOKEN=$(curl -sk -X POST "https://your-kamiwaza-instance/api/auth/token" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "username=admin&password=your-password" \
  | jq -r '.access_token')

# Step 2: Force a cache refresh
curl -sk -X POST "https://your-kamiwaza-instance/api/v1/apps/{app_id}/refresh" \
  -H "Authorization: Bearer ${TOKEN}"
```

General Troubleshooting Steps

When encountering issues, follow these diagnostic steps:

1. **Check Service Status**: Verify all Kamiwaza services are running
2. **Review Logs**: Check container logs for specific error messages
3. **Verify Resources**: Ensure sufficient CPU, RAM, and disk space
4. **Test Connectivity**: Verify network connectivity between components
5. **Restart Services**: Try stopping and restarting affected services
6. **Check Configuration**: Verify configuration files and environment variables

SpiceDB maintenance upgrade

> **UNRELEASED PROCEDURE**
>
> This draft applies to the first Kamiwaza release that makes SpiceDB the default relationship-based access control (ReBAC) backend. That release has not yet been assigned a version here. It does not describe the defaults or upgrade procedure of a previously released version. Confirm the selected release and its supported upgrade path with Kamiwaza before scheduling this change.

Draft release note

SpiceDB becomes the default authorization backend for installations with ReBAC enabled. The supported install and upgrade procedure requires a maintenance window with a service outage while authorization data is prepared and verified. The Core SQL database retains the stored relationships; SpiceDB evaluates graph permissions. Existing explicit backend selections remain configuration choices, and a graph outage does not automatically switch the default configuration to PostgreSQL.

Plan maintenance for subsequent graph-authoritative API replica starts too, including planned restarts and scale-out. Each starting replica repeats preparation and can pause authorization writes and sign-in requests that synchronize identity-provider data. This procedure does not promise upgrades or replica replacement without an outage.

Plan the maintenance window

Agree on the outage, the customer change owner, the Kamiwaza Support contact, and the point at which an incomplete upgrade must move to recovery. Keep customer traffic closed until every validation step below passes.

Rehearse the selected release against a representative copy of the site's data and configuration. Record time for backup and restore validation, any remaining image transfer or import, installation, authorization preparation, replica startup, and application checks. Reserve additional time for diagnosis and the agreed recovery procedure. Stage verified artifacts before the outage when the installation method supports it.

Sites using mostly the supplied ReBAC relationships have less custom authorization data to inspect and migrate. This is useful planning context, not a measured duration or guarantee. Tenant count, existing graph data, pending changes, resource limits, storage performance, and replica count affect the work. Concurrent replica starts serialize preparation; adding replicas can extend the window.

The proposed chart defaults provide failure limits, not expected completion times:

Boundary	Default limit	Meaning
Schema availability	900 seconds (15 minutes)	Wait for the required SpiceDB schema.
One cutover attempt	600 seconds (10 minutes)	Acquire writer exclusion, validate SQL data, recover pending work, reconcile, and verify.
SQL completion barrier and SpiceDB bootstrap Jobs	3,900 seconds (65 minutes)	Kubernetes execution deadline for the relevant release-managed Jobs.
Application Helm operation	4,200 seconds (70 minutes)	Installer wait limit; site overrides may differ.

Do not add these values to produce a maintenance estimate or treat a timeout as proof that a process stopped. Confirm the selected package's effective values and use the rehearsal to schedule the window. If a limit is insufficient, review the cause and all dependent limits with Kamiwaza before changing it.

Before maintenance

1. Confirm the exact source release, target release, installation method, chart values, image versions, cluster context, namespace, and desired replica counts. Use the selected release's supported installation procedure (see online documentation). Do not reuse an older installer or infer release inputs from this draft.
2. Record backend selections for every namespace. Preserve explicit overrides unless the upgrade plan deliberately changes them. Confirm whether SpiceDB is managed by the installation or supplied externally, and verify its credentials, datastore, connectivity, and CA trust for both application workloads and initialization Jobs. Keep credentials in the supported Secret mechanism.
3. Confirm with Kamiwaza Support whether the selected release's read-only SQL authorization preflight can read the installed database schema, and obtain the supported invocation. The incoming Core package requires compatible, initialized SQL tables. If SQL migration must happen first, plan the supported all-PostgreSQL preparation phase during maintenance, then run preflight before enabling graph authority. Otherwise, run it before the window. For an existing SpiceDB or shadow deployment, also run the complete graph identity preflight. Resolve incompatible tuples and identity formats before cutover; preserve the reports.
4. Review supported group grants, nested membership, mission grants, workroom sharing, and group clearance grants reported by preflight. Graph evaluation can make stored grants effective for members where the previous SQL evaluator did not expand them. A passing compatibility check does not establish unchanged access decisions. Identify representative allowed and denied users and resources for the final access checks.
5. Confirm a restorable backup and a recovery plan for the site's complete persistent state, including the Core database and any existing SpiceDB datastore. Record how the compatible prior software and configuration would be restored. Plan the final backup after writers are stopped so that the recovery point matches the upgrade boundary.

Maintenance procedure

1. Close traffic and stop competing work

Announce maintenance and use the site's approved ingress or access controls to stop new customer and internal client requests. Drain requests already in progress. Pause scheduled jobs, ingestion, automation, operator job triggers, and other background work that can change authorization data or launch services. Closing browser access alone does not stop these writers.

Using the release's supported maintenance controls, stop the older application replicas and every older graph or shadow projector. Verify that their SQL transactions and graph operations have completed and that their controllers or automation will not restart them during the transition. Keep PostgreSQL, SpiceDB, storage, and other dependencies needed by the installer available.

An older binary may write relationships without the new projection protocol. It must not remain active after graph authority is enabled. If the supported upgrade path requires old and new binaries to overlap, first use its explicit PostgreSQL preparation stage for all namespaces, with older graph and shadow projectors stopped. Enable graph authority only after the older binaries are gone and preflight passes. This staged configuration is planned migration work; it is not an automatic response to a SpiceDB failure.

Take and validate the final backup at this quiet boundary. Record the backup, configuration, preflight reports, and the maintenance start time.

2. Run the supported installer

Run the selected release's installer with the approved site configuration. Keep its initialization Jobs and new application resources enabled. Do not scale every workload to zero or use a configuration that prevents the installer from creating the new API replicas and bootstrap Jobs.

If the incoming preflight needs a newer SQL schema, first complete the supported preparation phase with all namespaces explicitly on PostgreSQL and no SpiceDB namespace overrides. Keep older graph and shadow projectors stopped. Run preflight against the initialized schema and resolve its findings before the graph-authoritative phase. Keep maintenance in effect across both phases.

Allow the complete installation sequence to finish, including SQL initialization, SpiceDB datastore migration, schema initialization, baseline seeding, and the final cutover after seeding. Check regular Jobs as well as Helm hooks; a healthy older graph service or a successful schema wait does not prove the migration Job succeeded.

Every new graph-authoritative API replica also prepares and verifies the graph before becoming ready. An API replica becoming ready before final seeding does not replace the final cutover check. Keep external traffic and paused background work closed throughout this sequence.

If any stage fails, keep maintenance in effect and follow Failure and recovery (see online documentation). Do not bypass the failed stage or repeat the installer without diagnosing the first failure.

3. Validate before reopening

Confirm all of the following against the selected release:

- The installer completed successfully, and the required SQL, datastore migration, schema, seed, and final cutover Jobs completed successfully.
- The intended API replicas are ready, workers and other authorization clients run the intended version and configuration, and no older writers or projectors remain active.

- Authorization reconciliation and readiness checks pass with no unresolved projection failures or missing projection receipts in the required graph scope. Runtime projection is active; a successful one-time cutover does not replace it.
- A representative administrator and standard user can sign in and use the resources their policies allow. Include the site's identity-provider synchronization configuration in these checks.
- A real policy-negative check denies a user access to a protected resource. A service-unavailable response is not evidence of a correct policy denial. Check membership and clearance cases identified during preflight.
- On a disposable resource with a suitable test user, create a grant through the normal application workflow, observe successful projection, and confirm that user's access. Revoke the grant, observe successful projection again, and confirm a policy denial from the same user's session. Retain the results and remove the test resource. This exercises ongoing projection after cutover.

Complete the ReBAC validation checklist (see online documentation), record the results and elapsed time, and obtain the change owner's confirmation to end maintenance. Reopen traffic and resume paused work in a controlled order, watching sign-in, authorization failures, and projection health.

Later restarts and scale-out

Preparation repeats on every graph-authoritative API replica start, including planned restarts and scale-out. Schedule those operations using the same maintenance boundary and readiness checks. Existing replicas serving requests do not make a concurrent preparation an operation without an outage.

An automatic or unplanned restart can begin preparation outside a scheduled window. It can pause authorization writes, delay synchronized sign-in, and require an outage while an operator restores a stable state. If preparation is interrupted or does not finish, keep or establish maintenance and use the recovery process below.

Failure and recovery

Keep traffic closed and background work paused. Retain the first failed installer output, Job status and logs, preflight reports, application startup logs, database diagnostics, and the selected release inputs. Work with Kamiwaza Support to identify the failed stage and affected tenant or relationship before retrying.

An interrupted process or unreachable host can leave PostgreSQL retaining an authorization lock until it detects the lost connection. A timeout or missing pod does not prove that a previously issued graph write has finished. Before any targeted cleanup of an orphaned database session, Support and the database operator must identify its originating process or host, stop or isolate that writer so it cannot resume graph writes, and establish that its issued graph operations have completed. Only then may they release that exact orphaned session. Do not terminate all database sessions or rely on a database timeout to make graph operations safe.

After correcting the cause and establishing exclusive ownership, run the selected release's full cutover recovery, then repeat all readiness and access checks before reopening. Do not directly reset failed projection rows, delete unknown graph relationships, skip identity-provider synchronization, or bypass schema and cutover checks to get the service online.

If recovery requires rollback, keep maintenance in effect and follow the previously agreed restore plan for compatible software, configuration, and persistent data. Merely selecting PostgreSQL after a graph failure is not a verified rollback. Do not reopen on a partial restore or an unverified backend change. An explicitly planned PostgreSQL rollback uses direct-tuple evaluation, so group-, mission-, and workroom-inherited access can change. Include those cases in the rollback access checks.

Ontology Graph Viewer Runbook

The **Graph** tab in a workroom shows a knowledge graph built from the workroom's sources. The Graph tab renders an empty or "being set up" state when the underlying ontology graph instance for the workroom is not yet running. This page describes what those states mean and how to recover from them.

If you reached this page from the **View runbook** button in the Graph tab, the workroom's graph instance is provisioned but not in a running state, or is still being provisioned and has exceeded the typical setup window.

What You May See

The Graph tab surfaces two empty states that link to this runbook:

- **"Knowledge graph is being set up"** — the workroom's graph instance is `pending`. The auto-provisioner deploys the graph backend in the background; typical provisioning takes around 30 seconds. The UI polls instance status for up to 60 seconds and then stops auto-refreshing.
- **"Graph instance idle"** — the workroom's graph instance was provisioned at some point but is not currently running. The empty state displays the instance status string (for example, `stopped`, `failed`, `unknown`).

For context, two adjacent Graph tab states that do not link here but appear elsewhere in this page:

- **"No ontology instance"** — shown before any provisioning has occurred for the workroom, with a "Create ontology instance" CTA gated on the admin role. This state is the post-recovery target of the Recreate the Instance (see online documentation) step below.
- **"Graph unavailable"** — the backend-error state, shown with a retry action and a correlation ID for support. If you saw this and were redirected to this runbook, include the correlation ID when you ask for help.

The first response to either of the runbook-linked states is to refresh the page. The tab re-queries instance status on reload, and a transient state may resolve on its own.

"Knowledge graph is being set up"

If this message persists for longer than a minute or two, the auto-provisioner is not completing. Check the underlying pods:

```
kubectl get pods -n kamiwaza -l extensions.kamiwaza.io/name=service-graphiti
```

The Kamiwaza extension operator labels graph backend pods with `extensions.kamiwaza.io/name=service-graphiti`. If the selector returns no pods, the operator has not yet created them — confirm the `service-graphiti` `KamiwazaExtension` resource is present in the namespace and reconciling without errors.

If the pods exist but are not `Running`, inspect them:

```
kubectl describe pod -n kamiwaza <pod>
kubectl logs -n kamiwaza <pod>
```

Common causes for a stuck `pending` state include:

- The image is still being pulled (check `describe` events).
- A required secret or config value is missing.
- The underlying graph database is starting up or failing its readiness probe.
- The pod cannot reach a dependency it needs to become ready.

"Graph instance idle"

The instance status string in the empty state indicates the broad failure mode. Use the same selector to find the pods:

```
kubectl get pods -n kamiwaza -l extensions.kamiwaza.io/name=service-graphiti \
  -L extensions.kamiwaza.io/deployment-id
```

`-L extensions.kamiwaza.io/deployment-id` surfaces the per-workroom deployment id as a column rather than printing the full label set. Each per-workroom graph instance is provisioned as a separate `KamiwazaExtension` and carries a unique `extensions.kamiwaza.io/deployment-id` label, which the operator also propagates to the owning Deployment — so the same label scopes both pod-level and Deployment-level commands later in this runbook. To list just the deployment-id values currently in the namespace:

```
kubectl get pods -n kamiwaza -l extensions.kamiwaza.io/name=service-graphiti \
  -o jsonpath='{range .items[*]}{.metadata.labels.extensions\.kamiwaza\.io/deployment-
id}{"\t"}{.metadata.name}{"\n"}{end}' \
  | sort -u
```

Cross-reference the listed deployment-ids against the affected workroom's `ontology_id` from the Graph tab status payload to pick the one to recover. The subsequent recovery steps use this value to scope actions to that single instance.

A `failed` status indicates the platform tried to reconcile the instance to running and surfaced a terminal error. A `stopped` status means the instance was explicitly stopped. An `unknown` status means the platform could not determine the instance's current state at all.

In each case, the next step is the same: inspect the pods and the operator's reconciliation events, identify the root cause, and start with the least destructive recovery — pod restart — before resorting to recreating the instance.

Recovering a Stuck Instance

Try a Pod Restart First

If the underlying pods are unhealthy but the instance record itself looks fine, restart the deployment before deleting anything. Scope the restart to the single workroom's instance using the `deployment-id` value you noted above — the namespace may host multiple Graphiti deployments, one per workroom, and restarting all of them would bounce unrelated workrooms:

```
kubectl rollout restart deployment -n kamiwaza -l extensions.kamiwaza.io/deployment-id=
<deployment-id>
kubectl rollout status deployment -n kamiwaza -l extensions.kamiwaza.io/deployment-id=
<deployment-id> --timeout=180s
```

Refresh the Graph tab once the rollout reports ready. If the instance returns to running, no further action is needed.

Recreate the Instance

If a pod restart does not recover the instance — for example, the instance is in a terminal `failed` state, or the underlying configuration that produced it is no longer correct — an administrator can delete the instance and let the platform's auto-provisioner deploy a fresh one the next time the workroom is opened.

> **CAUTION**
>
> Recreating a graph instance discards the graph data that was built up for the workroom. Sources will need to be re-ingested into the new instance before the Graph tab repopulates. Do not recreate an instance while a long-running ingestion is in flight unless you intend to redo it.

Issue the following call against the Kamiwaza API gateway with an admin bearer token:

```
curl -X DELETE "https://<your-domain>/api/context/ontologies/<ontology-id>" \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Workroom-ID: <workroom-id>"
```

`<your-domain>` is the host serving the Kamiwaza UI, and `<ontology-id>` is the workroom's current ontology id (see below for retrieval). The `X-Workroom-ID` header is optional per the OpenAPI spec but is the workroom-isolation header used by the surrounding API; include it on admin recovery calls so the operation stays scoped to the right workroom. The bearer token must belong to an account with administrative authority on the platform.

To retrieve the ontology id for a workroom, list its ontology instances via the same API. The endpoint accepts the same `X-Workroom-ID` header for scope:

```
curl "https://<your-domain>/api/context/ontologies/" \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Workroom-ID: <workroom-id>"
```

The response payload includes each instance's `id` (the `ontology_id` referenced above), `status`, and runtime endpoint. Alternatively, the same payload backs the Graph tab in the UI — operators with browser access can read it from the network panel by opening the affected workroom's Graph tab.

After the delete completes, reload the Graph tab in the workroom; the empty state will return to **"No ontology instance"** (described above), and the auto-provisioner will create a fresh instance when the next admin or eligible user opens the workroom.

What to Gather Before Reaching Out

If the steps above do not recover the Graph tab, share the following with Kamiwaza support or your platform team:

- **Workroom ID** — visible in the workroom URL.
- **Instance status string** — the value shown in the Graph tab empty state (for example, `pending`, `failed`).
- **Pod state** — output of `kubectl get pods -n kamiwaza -l extensions.kamiwaza.io/name=service-graphiti -o wide`, plus `kubectl describe` for any pod that is not `Running`.
- **Correlation ID** — if you reached the **Graph unavailable** error state, the correlation ID it displays.
- **Approximate time** the issue began.

See Help & Fixes (see online documentation) for the broader set of Kamiwaza support channels.

Kamiwaza SDK

Build against Kamiwaza using the Python SDK and the REST API reference.

- Use the sidebar to browse Python SDK services.
- Open the REST API Reference (see online documentation) for endpoint-level details.

REST API Reference

The offline build omits the interactive REST API reference to keep SDK navigation responsive when browsing directly from disk.

Use these options instead:

- Browse the Python SDK service docs in the sidebar for offline usage.
- Open the hosted API reference when you have network access: docs.kamiwaza.ai/sdk/api/ (see online documentation)

Activity Service

Overview

The Activity Service (`ActivityService`) provides comprehensive activity tracking and monitoring functionality for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/activity.py` , this service handles the tracking and retrieval of user and system activities across the platform.

Key Features

- Activity Tracking
- Recent Activity Retrieval
- Activity Filtering
- Timeline Management

Activity Management

Available Methods

- `get_recent_activity(limit: int = 50, skip: int = 0) -> List[Activity]` : Get recent activities

```
# Get recent activities
activities = client.activity.get_recent_activity(
    limit=10,  # Number of activities to retrieve
    skip=0     # Number of activities to skip
)

# Process activities
for activity in activities:
    print(f"Activity: {activity.type}")
    print(f"User: {activity.user_id}")
    print(f"Timestamp: {activity.timestamp}")
    print(f"Details: {activity.details}")
```

Integration with Other Services

The Activity Service works in conjunction with:

1. Authentication Service
    - For user identification

2. Lab Service
    - For lab activity tracking
3. Model Service
    - For model operation tracking
4. Cluster Service
    - For infrastructure events

Error Handling

The service includes built-in error handling for common scenarios:

```python
try:
    activities = client.activity.get_recent_activity()
except PermissionError:
    print("Insufficient permissions")
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Set appropriate activity limits
2. Implement activity filtering
3. Regular activity monitoring
4. Handle pagination properly
5. Process activities asynchronously
6. Implement proper error handling
7. Monitor activity patterns
8. Archive old activities

Performance Considerations

- Pagination impact on retrieval time
- Activity log size
- Query performance
- Storage requirements

Activity Types

The service tracks various types of activities:

1. User Actions
    - Login attempts
    - Resource creation
    - Configuration changes
2. System Events
    - Service status changes
    - Resource allocation
    - Error occurrences
3. Resource Operations
    - Model deployments
    - Lab creation/deletion
    - Data ingestion

App Garden Service

> **DEPRECATED:** The App Garden service (`client.apps`) is deprecated and will be removed in a future release. The Docker Compose-based App Garden is being replaced by Kubernetes CRD-based extensions. Use the **Extensions API** (`client.extensions`) instead.
>
> **Migration guide:**
>
> | Legacy (App Garden) | New (Extensions API) |
> |---|---|
> | `client.apps.list_deployments()` | `client.extensions.list_extensions()` |
> | `client.apps.deploy(template_id=..., name=...)` | `client.extensions.create_extension(reque` then `wait_for_extension(...)` |
> | `client.apps.get_deployment(deployment_id)` | `client.extensions.get_extension(name)` |
> | `client.apps.stop_deployment(deployment_id)` | `client.extensions.delete_extension(name)` |
>
> See the Extensions Service documentation (see online documentation) for full details.

The App Garden service allows you to deploy and manage containerized applications within the Kamiwaza platform. It provides a simple interface for deploying web applications, databases, and other services using Docker containers.

Overview

The App Garden is designed to make it easy to deploy applications alongside your AI workloads. It supports:

- Docker-based container deployment
- Pre-built application templates
- Environment variable configuration
- Automatic port allocation and routing
- Instance scaling and management

Authentication

Note: App Garden endpoints do not require authentication, making them accessible for quick deployments and testing.

Quick Start

```python
from kamiwaza_sdk import KamiwazaClient as kz

# Initialize client
client = kz("http://localhost:7777/api/")

# List available templates
templates = client.apps.list_templates()
for template in templates:
    print(f"{template.name} - {template.description}")

# Deploy an application from a template
deployment = client.apps.deploy(
    template_id=template.id,
    name="my-postgres-db",
    env_vars={"POSTGRES_PASSWORD": "mysecretpassword"}
)

print(f"Deployed: {deployment.name}")
print(f"Status: {deployment.status}")
```

Available Methods

Deployment Management

`deploy(template_id, name, **kwargs)`

Deploy a new application from a template.

Parameters:

- `template_id` (UUID): The ID of the template to deploy
- `name` (str): Name for your deployment
- `min_copies` (int, optional): Minimum instances to maintain (default: 1)
- `starting_copies` (int, optional): Initial number of instances (default: 1)
- `max_copies` (int, optional): Maximum instances allowed
- `env_vars` (dict, optional): Environment variables for the container

Returns: `AppDeployment` object with deployment details

`list_deployments()`

List all active application deployments.

Returns: List of `AppDeployment` objects

`get_deployment(deployment_id)`

Get details of a specific deployment.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: `AppDeployment` object

`stop_deployment(deployment_id)`

Stop and remove a deployment.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: Success message

Template Management

`list_templates()`

List all available application templates.

Returns: List of `AppTemplate` objects

`search_templates(query)`

Search for templates by name or description.

Parameters:

- `query` (str): Search term

Returns: List of matching `AppTemplate` objects

Instance Management

`list_instances(deployment_id)`

List all running instances of a deployment.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: List of `AppInstance` objects with host and port information

Image Management

`pull_images(deployment_id)`

Pull Docker images for a deployment (useful for pre-loading).

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: `ImagePullResult` with status information

Common Use Cases

Deploy a PostgreSQL Database

```
# Find PostgreSQL template
templates = client.apps.search_templates("postgres")
postgres_template = templates[0]

# Deploy with custom settings
deployment = client.apps.deploy(
    template_id=postgres_template.id,
    name="my-database",
    env_vars={
        "POSTGRES_PASSWORD": "secure_password",
        "POSTGRES_DB": "myapp"
    }
)

# Get connection details
instances = client.apps.list_instances(deployment.id)
for instance in instances:
    print(f"Connect to: {instance.host_name}:{instance.listen_port}")
```

Deploy a Web Application

```
# Deploy a web app with scaling
deployment = client.apps.deploy(
    template_id=web_template.id,
    name="my-web-app",
    min_copies=2,
    max_copies=5,
    env_vars={
        "API_KEY": "your_api_key",
        "DEBUG": "false"
    }
)
```

Monitor Deployments

```
# List all deployments
deployments = client.apps.list_deployments()

for deployment in deployments:
    print(f"\n{deployment.name}:")
    print(f"  Status: {deployment.status}")
    print(f"  Created: {deployment.created_at}")

    # Check instances
    instances = client.apps.list_instances(deployment.id)
    print(f"  Running instances: {len(instances)}")
```

Templates

Templates define the container configuration and requirements for applications. Each template includes:

- Docker image reference
- Default environment variables
- Port configurations
- Resource requirements
- Risk tier classification

Templates are verified by the Kamiwaza team and categorized by risk level for security.

Error Handling

```
try:
    deployment = client.apps.deploy(
        template_id=template_id,
        name="my-app"
    )
except Exception as e:
    print(f"Deployment failed: {e}")
```

Best Practices

1. **Use Templates**: Prefer templates over custom images for better security and reliability
2. **Environment Variables**: Store sensitive data like passwords in environment variables
3. **Scaling**: Set appropriate min/max copies based on your load requirements

4. **Monitoring**: Regularly check deployment status and instance health
5. **Cleanup**: Stop unused deployments to free resources

See Also

- **Extensions Service (see online documentation)** - The recommended replacement for App Garden (K8s-native extensions)
- Tool Service (see online documentation) - Deploy MCP Tool servers (also deprecated)

Authentication Service

The authentication helpers live in `kamiwaza_sdk/services/auth.py`. They wrap the Keycloak-backed auth gateway that fronts the Kamiwaza APIs.

> Base URL rule: set `base_url=https://<host>` (no `/auth` suffix). A quick preflight is `GET {base_url}/auth/ping` → 200. If base_url ends with `/auth`, SDK calls will double-prefix and fail.

Getting Started

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.authentication import UserPasswordAuthenticator

client = KamiwazaClient("https://localhost/api")

# Option 1 – Personal Access Token (recommended for automation)
# export KAMIWAZA_API_KEY=<pat>
# client automatically loads the token from the environment.

# Option 2 – username/password (interactive or CLI bootstrap)
client.authenticator = UserPasswordAuthenticator("admin", "kamiwaza", client.auth)

# Who am I?
me = client.auth.get_current_user()
print(me.username, me.roles)
```

> **Tip:** the SDK reads `KAMIWAZA_API_KEY` automatically. For federated users, generate a PAT in the UI (or with `client.auth.create_pat`) and drop it in the environment before instantiating the client.

> **Note:** `/auth/users/me` depends on Keycloak token introspection. In lite environments without Keycloak configured it may respond with 401; the token is still usable with ForwardAuth and PAT endpoints in that mode.

Tokens & Logout

```python
# Perform a password grant and capture the refresh token
token = client.auth.login_with_password("admin", "kamiwaza")

# Explicit refresh using the refresh token value
new_token = client.auth.refresh_access_token(token.refresh_token)

# Coordinated logout clears cookies and returns front-channel redirect metadata
logout = client.auth.logout()
print(logout.front_channel_logout_url)
```

`UserPasswordAuthenticator` handles this dance automatically: it refreshes when the token is close to expiring, keeps the refresh token, and falls back to a password grant if the refresh is rejected.

If a selected workroom is deleted while a password-authenticated session is still scoped to it, `client.workrooms.leave()` clears the stale in-memory credentials and session access-token state, then retries once with a fresh password grant. The shared on-disk token remains available during that attempt; a successful grant replaces it, while an exhausted grant clears the stale scoped token. PAT and API-key credentials are fixed: the SDK never clears or retries them as refreshable sessions. Custom authenticators may opt into the same bounded recovery by returning `True` from `invalidate_session(session) -> bool` after invalidating their session credentials.

Personal Access Tokens

```python
from kamiwaza_sdk.schemas.auth import PATCreate

# Create a PAT – token only appears once in the response
pat = client.auth.create_pat(PATCreate(name="local-cli", ttl_seconds=86400))
print(pat.token)

# List and revoke
pat_list = client.auth.list_pats()
client.auth.revoke_pat(pat_list.pats[0].jti)
```

Use PATs for long-lived automation. When a PAT is set in `KAMIWAZA_API_KEY`, the client automatically authenticates without any extra code.

Ingress authorization headers

Platform-managed ingress authorizes requests before they reach workloads such as App Garden apps. The SDK's legacy-named helper calls `/auth/validate` and parses the platform-attested identity header bundle returned by that endpoint.

```
headers = client.auth.forward_auth_headers()
print(headers.user_id, headers.user_roles)
print(headers.signature, headers.signature_ts)
```

A helper also surfaces the JWKS document for verifying PATs locally:

```
jwks = client.auth.get_jwks()
print(jwks.keys[0].kid)
```

Identity Providers & Local Users

Administrative helpers for IdP lifecycle, session purges, and local-user management mirror the REST surface:

```
from kamiwaza_sdk.schemas.auth import (
    RegisterIdPRequest,
    LocalUserCreateRequest,
    SessionPurgeRequest,
)

client.auth.list_identity_providers()
client.auth.purge_sessions(SessionPurgeRequest(
    tenant_id="__default__",
    subject_id="5ac2cafa-1fbd-4b66-aba1-40b6c023612c",
))

client.auth.create_local_user(LocalUserCreateRequest(
    username="new-admin",
    email="new-admin@example.com",
    password="changeme!",
    roles=["admin", "user", "viewer", "guest"],
))
```

These calls require an authenticated admin role and are intended for operational tooling rather than end-user flows. In Auth-on deployments, `create_local_user` provisions the Keycloak identity (transactional, rolls back on Keycloak failure), and `reset_user_password` updates Keycloak only (Keycloak is authoritative for login). In Auth-off mode, these calls remain DB-only.

Authorization Service (ReBAC)

Client helpers for the relationship-based access control surface exposed under `/auth/*` in the Kamiwaza API. Methods live in `kamiwaza_sdk/services/authz.py`.

Usage

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.authz import (
    CheckRequest,
    RelationshipTuple,
    SubjectModel,
    ObjectModel,
)

client = KamiwazaClient("https://localhost/api")

# Upsert a tuple (admin credentials required)
tuple_payload = RelationshipTuple(
    subject=SubjectModel(namespace="user", id="5ac2cafa-1fbd-4b66-aba1-40b6c023612c"),
    relation="owns",
    object=ObjectModel(namespace="dataset", id="urn:dataset:foo"),
)
client.authz.upsert_tuple(tuple_payload)

# Check access (resolves tenant from the authenticated token)
check = client.authz.check_access(
    CheckRequest(
        subject=tuple_payload.subject,
        relation="owns",
        object=tuple_payload.object,
    )
)
print(check.allow, check.decision_id)
```

Tuple delete helpers mirror the REST API:

- `client.authz.delete_tuple(...)`
- `client.authz.delete_object(...)`

Pass `tenant_id="__default__"` when you need to override the tenant that will own the tuple; otherwise it is inferred from the caller's token.

Catalog Service

Overview

The Catalog service exposes three focused clients that mirror the server-side routers:

Attribute	Description
`client.catalog.datasets`	Dataset CRUD, schema management, and URN helpers
`client.catalog.containers`	Container lifecycle and dataset membership
`client.catalog.secrets`	Secret registration/listing tied to corpuser owners

Each client accepts/returns the Pydantic models defined in `kamiwaza_sdk/schemas/catalog.py`, so responses are thread-safe and easy to validate.

Datasets

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.catalog import DatasetCreate, DatasetUpdate

client = KamiwazaClient("https://localhost/api", api_key="...shh...")

# Create and immediately fetch the dataset metadata
payload = DatasetCreate(
    name="/var/data/visitors.parquet",
    platform="s3",
    environment="PROD",
    description="Daily visitor export",
    properties={"region": "us-east-1", "endpoint": "http://localhost:19100"},
    tags=["pii", "bronze"],
)
dataset_urn = client.catalog.datasets.create(payload)
dataset = client.catalog.datasets.get(dataset_urn)

# List only datasets the receiver authorizes for this federated identity.
remote_datasets = client.catalog.datasets.list(target_cluster="receiver-name")

# Update tags/properties via URN
client.catalog.datasets.update(
    dataset_urn,
    DatasetUpdate(tags=["pii", "silver"], properties={"region": "us-east-1"}),
)

# Fetch or update the logical schema
schema = client.catalog.datasets.get_schema(dataset_urn)
client.catalog.datasets.update_schema(dataset_urn, schema)

# Convenience wrappers for legacy code that previously called
# `client.catalog.create_dataset(...)` still work and delegate to the dataset client.
```

Containers

```python
from kamiwaza_sdk.schemas.catalog import ContainerCreate, ContainerUpdate

containers = client.catalog.containers
container_urn = containers.create(ContainerCreate(name="rag-demo",
platform="kamiwaza"))
containers.add_dataset(container_urn, dataset_urn)

# Update metadata or re-parent the container
containers.update(container_urn, ContainerUpdate(description="Demo assets"))

# Remove membership or delete the container
containers.remove_dataset(container_urn, dataset_urn)
containers.delete(container_urn)
```

All URNs can be percent-encoded for the `/v2/{urn}` endpoints with `client.catalog.encode_urn(...)` (or `DatasetClient.encode_path_urn` / `ContainerClient.encode_path_urn`).

Secrets

```
from kamiwaza_sdk.schemas.catalog import SecretCreate

secrets = client.catalog.secrets
secret_urn = secrets.create(
    SecretCreate(
        name="minio/rag-demo",
        value="
{'aws_access_key_id':'minioadmin','aws_secret_access_key':'minioadmin'}",
        owner="urn:li:corpuser:demo",
        description="Demo MinIO credentials",
    ),
    clobber=True,  # overwrite existing entry if present
)

secret = secrets.get(secret_urn)
for visible in secrets.list():
    print(visible.urn, visible.owner)

secrets.delete(secret_urn)
```

Secret URNs are generated by the server (often DataHub-style such as `urn:li:dataHubSecret:<name>`). Treat them as opaque identifiers—capture the value returned by `create(...)` and pass it verbatim to subsequent `get` / `delete` calls rather than constructing prefixes client-side.

The SDK now calls the `/catalog/secrets/v2/{secret_urn}` path with the raw URN, so you shouldn't need to percent-encode or reshape it yourself; keep the string exactly as the server returned.

Diagnostics

Both the catalog and auth services now expose lightweight health endpoints to simplify readiness checks:

```
client.auth.health()        # -> {"status": "ok", ...}
client.auth.metadata()      # -> {"version": "...", ...}
client.catalog.health()     # -> {"status": "ok"}
client.catalog.metadata()   # -> {... service banner ...}
```

These helpers pair with the improved `VectorDBUnavailableError` so SDK consumers can reason about platform readiness before kicking off ingestion or retrieval jobs.

Cluster Service

Overview

The Cluster Service (`ClusterService`) provides comprehensive cluster and infrastructure management for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/cluster.py`, this service handles location management, cluster operations, node management, and hardware configuration.

Key Features

- Location Management
- Cluster Operations
- Node Management
- Hardware Configuration
- Runtime Configuration
- Hostname Management

Location Management

Available Methods

- `create_location(location: CreateLocation) -> Location`: Create new location
- `update_location(location_id: UUID, location: UpdateLocation) -> Location`: Update location
- `get_location(location_id: UUID) -> Location`: Get location info
- `list_locations() -> List[Location]`: List all locations

```python
# Create new location
location = client.cluster.create_location(CreateLocation(
    name="us-west",
    provider="aws",
    region="us-west-2"
))

# Update location
updated = client.cluster.update_location(
    location_id=location.id,
    location=UpdateLocation(name="us-west-prod")
)

# Get location details
location = client.cluster.get_location(location_id)

# List all locations
locations = client.cluster.list_locations()
```

Cluster Management

Available Methods

- `create_cluster(cluster: CreateCluster) -> Cluster`: Create new cluster
- `get_cluster(cluster_id: UUID) -> Cluster`: Get cluster info
- `list_clusters() -> List[Cluster]`: List all clusters
- `get_hostname() -> str`: Get cluster hostname

```python
# Create new cluster
cluster = client.cluster.create_cluster(CreateCluster(
    name="training-cluster",
    location_id=location_id,
    node_count=3
))

# Get cluster info
cluster = client.cluster.get_cluster(cluster_id)

# List clusters
clusters = client.cluster.list_clusters()

# Get hostname
hostname = client.cluster.get_hostname()
```

Node Management

Available Methods

- `get_node_by_id(node_id: UUID) -> Node`: Get node info
- `get_running_nodes() -> List[Node]`: List running nodes
- `list_nodes() -> List[Node]`: List all nodes

```
# Get node details
node = client.cluster.get_node_by_id(node_id)

# List running nodes
running_nodes = client.cluster.get_running_nodes()

# List all nodes
all_nodes = client.cluster.list_nodes()
```

Hardware Management

Available Methods

- `create_hardware(hardware: CreateHardware) -> Hardware`: Create hardware entry
- `get_hardware(hardware_id: UUID) -> Hardware`: Get hardware info
- `list_hardware() -> List[Hardware]`: List hardware entries
- `get_runtime_config() -> RuntimeConfig`: Get runtime configuration

```
# Create hardware entry
hardware = client.cluster.create_hardware(CreateHardware(
    name="gpu-node",
    gpu_count=4,
    gpu_type="nvidia-a100"
))

# Get hardware info
hardware = client.cluster.get_hardware(hardware_id)

# List hardware
hardware_list = client.cluster.list_hardware()

# Get runtime config
config = client.cluster.get_runtime_config()
```

Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    cluster = client.cluster.create_cluster(cluster_config)
except LocationNotFoundError:
    print("Location not found")
except ResourceError as e:
    print(f"Resource allocation failed: {e}")
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Validate location existence before cluster creation
2. Monitor node health regularly
3. Use appropriate hardware configurations
4. Implement proper error handling
5. Clean up unused resources
6. Consider resource limits
7. Monitor cluster performance
8. Use meaningful naming conventions

Performance Considerations

- Node count affects cluster performance
- Hardware configuration impacts resource availability
- Location selection influences latency
- Runtime configuration affects resource utilization

Connectors

`client.connectors` covers two distinct audiences:

- **Instance administration** (admin-scoped, cluster-wide): register a connector type, create/update/delete a configured instance, subscribe an out-of-core connector.
- **Connector surfaces** (member-scoped, per workroom): discover which connectors the acting user may use, verify their connection, and browse, search, and fetch content from a connector's read surfaces.

The per-user OAuth ceremony that connects a member's own provider account is an interactive platform flow and is deliberately not wrapped by the SDK. The platform brokers provider tokens itself, so no method here accepts, returns, or stores a provider credential.

Instance administration

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.connectors import ConnectorCreate

client = KamiwazaClient(base_url="https://kamiwaza.example/api")

catalog = client.connectors.list_catalog()  # .items and .source_status (admin)
client.connectors.list()              # every registered connector (admin view)
client.connectors.list_available()    # enabled connectors, user-safe metadata
client.connectors.get(connector_id)

client.connectors.create(
    ConnectorCreate(
        name="Microsoft 365",
        connector_type="m365",
        config={"tenant_id": "…", "client_id": "…"},
        scopes=["Files.Read.All"],
    )
)
```

`connector_type` is an open string resolved against the published catalog, and `config` is opaque to the SDK — the platform validates it against the connector's manifest. New connector types therefore need no SDK change.

See also `register_type` (publish a connector type to the cluster catalog), `subscribe` (attach an out-of-core connector by manifest + endpoint), `update`, and `delete`.

Connector surfaces

Surface operations are scoped to a **workroom** and execute as the calling member: the platform re-evaluates the user's provider ACLs on every request. A connector absent from the surface catalog is not reachable through any other method on this service — the catalog is the authoritative discovery call.

Discovery

```
catalog = client.connectors.list_surface_catalog(workroom_id)

for entry in catalog:
    entry.label            # "Microsoft 365"
    entry.connected        # did this member connect their own account?
    entry.ready_surfaces()      # surfaces registered and healthy in the workroom
    entry.searchable_surfaces() # …of those, the ones supporting full-text search
```

`list_surface_catalog(workroom_id, connected_only=True)` narrows the same call to instances the member has actually connected. The platform returns unconnected instances too — knowing a connector exists is what lets a UI offer to connect it.

A surface is usable only when its `workroom_state` is `ready`; the `ConnectorSurfaceCapability.ready` property encodes that, and anything else (including `unknown`) is treated as unusable rather than optimistically tried.

Verification

```
result = client.connectors.verify_connection(connector_id)
if result.available:
    ...
```

`verify_connection` probes the provider under the member's stored credential and reports per-capability results. **It is the one write on this surface**: the platform persists the health it classifies, so a verify can move the member's stored connection into a degraded or reauth-required state, and it issues live third-party calls. Call it deliberately — on connect, on an explicit user action, or after a surface call already failed — not on a timer and not fanned out across a catalog. Gate on `available` — the fail-closed verdict that requires both a usable `connection_status` and a passing check — rather than reading `status` directly. `connection_usable` and `checks_passed` expose the two halves separately when a caller needs to explain *why* access was refused. An ambiguous result (unknown status, `partial` with no successful check, a connection needing reauth) resolves to `available is False`.

Browse and search

```
from kamiwaza_sdk.schemas.connector_surfaces import (
    ConnectorBrowseRequest,
    ConnectorSearchRequest,
    ConnectorSurfaceRef,
)

# The workroom and connector instance always travel together.
ref = ConnectorSurfaceRef(workroom_id=workroom_id, connector_id=connector_id)

page = client.connectors.browse_surface(
    ref,
    ConnectorBrowseRequest(surface="files", container_id="folder-1", page_size=50),
)

page.items            # list[ConnectorNode]
page.next_page_token  # opaque; pass back as page_token for the next page
page.has_more

hits = client.connectors.search_surface(
    ref,
    ConnectorSearchRequest(surface="files", query="quarterly report"),
)
```

`page_size` is clamped into the platform's accepted 1–200 range rather than rejected. Search is only accepted on surfaces that declare `search_supported`; check `searchable_surfaces()` first instead of discovering it as a 501.

To walk several pages, pass the previous page's `next_page_token` back in and stop on your own terms — the SDK deliberately ships no exhaustive-walk helper, because a bounded one would have to choose between hanging on a large surface and silently truncating it, and only the caller knows which is acceptable:

```
request = ConnectorBrowseRequest(surface="files")
while True:
    page = client.connectors.browse_surface(ref, request)
    handle(page.items)
    if not page.has_more:
        break
    request = request.model_copy(update={"page_token": page.next_page_token})
```

Content fetch

```python
from kamiwaza_sdk.schemas.connector_surfaces import ConnectorContentRequest

node = page.items[0]
content = client.connectors.fetch_surface_content(
    ref,
    node.id,
    ConnectorContentRequest.from_node(node),
)

if content.is_text:
    body = content.text(limit=100_000)
else:
    blob = content.content          # raw bytes, never silently decoded
```

`ConnectorContentRequest.from_node` copies the node's surface and its `content_handle.query` provider locator, so callers never hand-assemble provider identifiers. A `drive_id` or `mime_type` that arrives only inside the locator is promoted into the declared parameter rather than dropped — connectors do ship their export `mime_type` that way. An explicitly-set field always wins over a locator entry of the same name, and `surface` is never accepted as a locator key.

Any *other* locator key is still sent, but the platform's content route declares only `surface`, `drive_id` and `mime_type`, so an undeclared key is discarded server-side today. Passing one is harmless and costs nothing if the route later declares it — it is not a forward-compatibility guarantee.

Node ids are opaque and are percent-encoded on the way out. When the platform sends a `Content-Disposition`, its filename wins over the caller's guess.

A missing content type counts as text, matching how the platform serves provider payloads that declare no type, and the check is case-insensitive. When the platform sends a `Content-Disposition`, its filename wins over the caller's guess — but only after being rejected for path separators, since that header is the third-party provider's text and callers join it onto a directory. RFC 5987's `filename*` is preferred and decoded; a plain `filename` is used verbatim.

Failure behavior

Surface methods let `APIError` propagate with its `status_code` intact instead of collapsing failures into one exception, because callers must distinguish a permanent denial from a transient outage:

Status	Meaning
400	Invalid request; a connector id this workroom does not resolve; an unknown surface; **or a surface whose workroom registration is not** `ready`
403	Not permitted for this workroom / member
404	Connector or node not found
409	The member has no connection to this connector — send them to the catalog entry's `reauth` deep links, not a retry
413	Content exceeds the platform's size cap (permanent for that node)
429	Provider rate limit — transient, back off
501	**Ambiguous:** either the surface declares no such operation (permanent) *or* the connector is not deployed right now (transient)
502	Upstream provider error
503	Connector not deployed; verification impossible

Two of these are easy to get wrong:

- **A surface that is not ready is a 400, not a 409.** The platform raises it as a request error alongside an unknown surface name. Routing it to the reconnect flow sends the member somewhere that cannot help them.
- **501 carries two meanings and the platform does not distinguish them.** Retry once with backoff before recording a surface as permanently unsupported, otherwise a connector rollout looks like a missing capability.

401 never reaches you as an `APIError` — and it covers two different problems. The platform returns it both when *your* session credential is rejected and when the *connector's* provider token needs reauthorization (`TokenRefreshException`), which is a permanent, member-actionable condition. The client intercepts every 401 before service code runs. Refreshable authenticators may refresh the credential and retry once: PATs do not retry. It then raises `AuthenticationError`, which descends from `KamiwazaError`, *not* `APIError`, and, with the built-in PAT and password authenticators, carries `status_code=401`. (If a refresh after the 401 fails, the status is the refresh outcome's, so a disabled account is not reported as a 401. A shared-IdP or custom authenticator whose refresh fails raises its own `AuthenticationError` unchanged.) The message names the request as `METHOD /path` (query string removed) and, when the platform returns a recognized reason, a fixed label such as `Token expired`. So this does not fire:

```
except APIError as exc:
    if exc.status_code == 401:   # unreachable on these methods
        ...
```

Catch `AuthenticationError` separately. When the client retries before raising, an expired credential also costs a duplicate content download. Both 401 causes still arrive as the same exception type with the same status. The fixed reason label is the only difference: the platform's connector reauthorization replies map to

labels ending in "the user must reconnect" (or, for a provider 401, `Session expired. Please reconnect.`), while a rejected session credential maps to labels such as `Token expired` or `Invalid token`. There is no structured field for this yet, so branch on type and status first and treat the label as advisory.

Timeouts

Every surface method takes a `timeout` override. Defaults are bounded: 15s for the catalog (a local metadata read) and 30s for verification, browse, search, and content fetch (all of which round-trip to a third-party provider).

Context Service

Overview

The Context Service (`ContextService`, accessed via `client.context`) is the SDK wrapper for the Kamiwaza Context Service. Located in `kamiwaza_sdk/services/context.py`, it manages workroom-scoped vector databases, ontologies (knowledge graphs), ingestion pipelines, and retrieval against the documents and knowledge held inside a **Workroom**.

Every Context Service resource lives inside exactly one workroom. The SDK targets a workroom by passing `workroom_id` to a method (sent to the server as the `X-Workroom-ID` header). Some methods make `workroom_id` a **required** keyword-only argument (e.g. `list_collections`, `create_pipeline_job`, `search`, `retrieve`, `upload_file`); others accept it optionally, and when it is omitted the server resolves the caller's default workroom.

For PAT/API-key automation that makes several calls against the same workroom, derive a local scoped client instead of calling `workrooms.enter()`:

```
with client.workroom_scope(my_workroom_id) as scoped:
    db = scoped.context.create_vectordb(name="project-vdb", engine="milvus")
    scoped.context.insert_vectors(
        db["id"],
        collection_name="project_docs",
        vectors=[embedding],
        metadata=[{"source": "seed"}],
    )
```

The scoped client only adds the explicit workroom header on SDK requests; it does not change the parent client or mutate server-side selected-session state. It is not a client-side security boundary; the server must still authorize the caller for the requested workroom on every request.

Workrooms and the Global Workroom

A **Workroom** is the collaboration and isolation boundary for context: vector collections, ontologies, ingested files, and pipeline jobs all belong to a workroom, and access is scoped to that workroom's members.

The **Global Workroom** is a special, well-known workroom (the all-`f` sentinel `ffffffff-ffff-ffff-ffff-ffffffffffff`, exposed as `ContextService.DEFAULT_WORKROOM_ID`). It holds the platform's **shared, read-only catalog** of context — knowledge that is visible to everyone but owned by no single member.

The Global Workroom is read-only for tenant writes

> **Key semantics:** Any direct write that targets the Global Workroom is rejected by the server with **HTTP 403** and a body of `Global Workroom is read-only for <operation>` (for example `Global Workroom is read-only for ontology creation`). This is **intentional and by design**, not a bug — the Global Workroom is a shared catalog, so it is populated only by the platform's own ingestion paths, never by ad-hoc tenant writes.

Reads against the Global Workroom are always allowed (results are requester-scoped where appropriate). Writes are blocked. The table below lists the **server-side policy categories** — some (e.g. pipeline retry/rerun/delete, workroom archive/restore/purge) are enforced by the Context Service but are not all surfaced as `client.context` methods; the SDK exposes a subset (see the method lists below).

Operation category	Against a normal workroom	Against the Global Workroom
List / get / query / search / retrieve (reads)	✅ allowed	✅ allowed
VectorDB create / update / delete / insert	✅ allowed	❌ 403 read-only
Ontology create / delete / add_knowledge / add_entity / delete_group	✅ allowed	❌ 403 read-only
Collection create / delete, chunk indexing / deletion	✅ allowed	❌ 403 read-only
Pipeline job create / cancel / retry / rerun / delete	✅ allowed	❌ 403 read-only
Raw file upload	✅ allowed	❌ 403 read-only
Workroom archive / restore / purge (lifecycle)	✅ allowed	❌ 403 read-only

> **Note on `*_global` helpers.** `query_vectors_global()` is a **read** against the shared catalog and works for any caller. `insert_vectors_global()` targets the same shared catalog and is reserved for the platform's connector-runtime ingestion path — a direct tenant call will hit the same 403 read-only policy. To store your own vectors, create or use a normal (non-global) workroom.

If you are migrating tests or code that assumed the Global Workroom was writable: point the write at a workroom you own (pass its `workroom_id`), and only read from the Global Workroom.

VectorDB

Available Methods

- `list_vectordbs(*, workroom_id=None)`
- `get_vectordb(vectordb_id, *, workroom_id=None)`
- `create_vectordb(*, workroom_id=None, ...)`
- `update_vectordb(vectordb_id, *, workroom_id=None, ...)`

- `scale_vectordb(vectordb_id, *, workroom_id=None, ...)`
- `delete_vectordb(vectordb_id, *, workroom_id=None)`
- `query_vectors(vectordb_id, *, collection_name, vectors, limit=10, workroom_id=None)`
- `query_vectors_global(*, vectordb_id, collection_name, vectors, limit=10)` — read the shared catalog
- `insert_vectors(vectordb_id, *, collection_name, vectors, metadata, workroom_id=None)`
- `insert_vectors_global(...)` — connector-runtime catalog writes only (see note above)

```python
# db_id, embedding, and my_workroom_id are values you supply.

# Read from the shared Global catalog (allowed)
hits = client.context.query_vectors_global(
    vectordb_id=db_id,
    collection_name="shared-docs",
    vectors=[embedding],
    limit=5,
)

# Write to a workroom you own (allowed) — NOT the Global Workroom
client.context.insert_vectors(
    db_id,
    collection_name="my-docs",
    vectors=[embedding],
    metadata=[{"source": "doc1"}],
    workroom_id=my_workroom_id,
)
```

Ontology (Knowledge Graph)

Available Methods

- `list_ontologies(*, workroom_id=None)` / `get_ontology(...)`
- `create_ontology(...)` / `delete_ontology(...)`
- `add_knowledge(...)` / `add_entity(...)` / `delete_group(...)`
- `search_knowledge(...)` / `get_memory(...)` / `get_episodes(...)`
- `ontology_health(...)`

Reads (`list_*`, `get_*`, `search_knowledge`, `get_memory`, `get_episodes`, `ontology_health`) work against any workroom including Global. Writes (`create_ontology`, `delete_ontology`, `add_knowledge`, `add_entity`, `delete_group`) are rejected on the Global Workroom.

Collections, Pipelines, and Retrieval

Available Methods

- `list_collections(...)` / `create_collection(...)` / `get_collection(...)` / `delete_collection(...)`
- `create_pipeline_job(...)` / `list_pipeline_jobs(...)` / `get_pipeline_job(...)`
- `get_supported_file_types()`
- `search(...)` / `retrieve(...)` / `upload_file(...)`

Pipeline cancel vs. delete

The SDK exposes two distinct teardown verbs for a pipeline job:

- `cancel_pipeline_job(...)` — **graceful** cancel (`POST /context/pipelines/{job_id}/cancel`). Stops the job but **preserves** its recorded history; the job stays fetchable via `get_pipeline_job(...)` .
- `delete_pipeline_job(...)` — **destructive** delete (`DELETE /context/pipelines/{job_id}`). Cancels a pending/running job and then removes the job **and** its history.

> **Breaking change:** `cancel_pipeline_job(...)` previously mapped to the destructive `DELETE` route. It now maps to the graceful cancel route, and the destructive behavior moved to the new `delete_pipeline_job(...)`. Update any caller that relied on the old `cancel_pipeline_job` to hard-delete a job.

Provider-neutral source import and replay

For Kaizen / import-shell automation, the SDK wraps the provider-neutral source-import and replay surface:

- `get_import_options(...)` / `evaluate_import_options(...)` — aggregate and validate import options for selected source descriptors.
- `create_source_import_job(...)` — start a provider-neutral source-import job.
- `list_import_items(...)` — workroom-wide source-import inventory/history.
- `rerun_import_items(...)` — rerun selected inventory items by recorded source descriptor.
- `list_pipeline_job_items(...)` — per-item statuses for one job.
- `retry_pipeline_job(...)` — retry failed/incomplete items from a replayable import job.
- `rerun_pipeline_job(...)` — rerun all recorded source descriptors from a prior import job.

Collection/pipeline/file **writes** follow the same rule: allowed in a normal workroom, `403` on the Global Workroom.

Raw-file Object Storage

The SDK wraps the workroom-scoped raw-file object-storage CRUD surface (`/context/storage/raw`). These methods are keyword-only and take an explicit `workroom_id` (sent as the `X-Workroom-ID` header); they return raw `dict` records rather than typed models.

Available Methods

- `store_raw_file(*, workroom_id, filename, content, ...)` — store a raw file directly into workroom object storage (`POST /context/storage/raw`). `content` accepts `bytes` or a `str` (UTF-8 encoded) and is base64-encoded on the wire for you. Optional `content_type`, `source_urn`, `source_kind`, `source_ref`, and `metadata` are forwarded only when set.
- `list_raw_files(*, workroom_id, ...)` — list stored raw files (`GET /context/storage/raw`), returning the `{"items": [...], "count": N}` shape. Supports `source_urn` / `job_id` / `connector_id` filters, `limit` / `offset` paging, and `include_markings=True` to attach aggregated security markings.
- `get_raw_file(file_id, *, workroom_id, ...)` — fetch one raw-file record (`GET /context/storage/raw/{file_id}`). Set `include_download_url=True` for a temporary presigned download URL (when S3 metadata exists); `expires_seconds` (30–3600) overrides its TTL.
- `update_raw_file(file_id, *, workroom_id, content, if_match=None)` — replace the plain-text body of an editable raw file (`PUT /context/storage/raw/{file_id}`). Pass the file's `updated_at` token from a prior response as `if_match` for optimistic concurrency; a stale token returns `409` with the current token in the error detail.

```
# my_workroom_id is a workroom you own.
stored = client.context.store_raw_file(
    workroom_id=my_workroom_id,
    filename="notes.txt",
    content="hello raw storage",
    content_type="text/plain",
)
file_id = stored["id"]

current = client.context.get_raw_file(file_id, workroom_id=my_workroom_id)
client.context.update_raw_file(
    file_id,
    workroom_id=my_workroom_id,
    content="edited body",
    if_match=current["updated_at"],  # optimistic concurrency
)
```

OmniParse Instances

The SDK wraps the workroom-scoped OmniParse instance lifecycle CRUD surface (`/context/omniparses`). OmniParse instances are the document-parsing runtimes that ingest pipelines depend on; wrapping them lets callers provision and manage OmniParse from automation rather than relying only on implicit lazy provisioning. These methods are keyword-only, take an explicit `workroom_id` (sent as the `X-Workroom-ID` header), and return raw `dict` / `list[dict]` records rather than typed models.

Available Methods

- `list_omniparses(*, workroom_id)` — list OmniParse instances for a workroom (`GET /context/omniparses`), returning a list of instance records.
- `get_omniparse(omniparse_id, *, workroom_id)` — fetch one instance by id (`GET /context/omniparses/{omniparse_id}`).

- `create_omniparse(*, name, workroom_id, template_name="tool-omniparse", config=None)` — provision an OmniParse runtime instance (`POST /context/omniparses`). `template_name` selects the App Garden tool template; `config` carries optional OmniParse environment configuration and is forwarded only when set.
- `update_omniparse(omniparse_id, *, workroom_id, config=None)` — update an instance's environment configuration (`PUT /context/omniparses/{omniparse_id}`).
- `delete_omniparse(omniparse_id, *, workroom_id)` — delete an instance (`DELETE /context/omniparses/{omniparse_id}`).

```
# my_workroom_id is a workroom you own.
instance = client.context.create_omniparse(
    name="docs-parser",
    workroom_id=my_workroom_id,
    config={"replicas": 1},
)
instance_id = instance["id"]

client.context.update_omniparse(
    instance_id,
    workroom_id=my_workroom_id,
    config={"replicas": 2},
)
client.context.delete_omniparse(instance_id, workroom_id=my_workroom_id)
```

Global Settings, Document Download, and Audio Readiness

These cover platform-wide Context configuration, presigned download of an original document, and an audio-ingest preflight probe.

Available Methods

- `get_global_settings()` — read platform-wide Context global settings (`GET /context/global-settings`). **Platform-scoped (ContextAdmin)**, not workroom-scoped — no `X-Workroom-ID` header is sent.
- `update_global_settings(*, omniparse=None, reason=None)` — patch the global settings (`PATCH /context/global-settings`). `omniparse` is a partial settings dict (e.g. `{"force_insecure_model_ssl": True}`); `reason` is an optional audit annotation. Only the provided fields change. Also ContextAdmin-scoped.
- `get_document_download_url(source_urn, *, workroom_id)` — get a presigned S3 download URL for an original document by source URN (`GET /context/documents/{source_urn}`). Returns `download_url` plus `filename` / `content_hash` / `source_urn` . Requires S3 document storage to be enabled server-side.
- `get_audio_readiness(*, workroom_id, mime_type=None, filename=None)` — probe whether a workroom is ready to ingest audio (`GET /context/audio-readiness`). Optional `mime_type` (e.g. `audio/aiff`) and `filename` refine the check. Returns the readiness verdict (`ready` / `code` / `message` / `remediation` / `details`). This GET never side-effect-provisions an OmniParse instance.

```
# Platform-scoped admin config (no workroom).
settings = client.context.get_global_settings()
client.context.update_global_settings(
    omniparse={"force_insecure_model_ssl": False},
    reason="rotate certs",
)

# Workroom-scoped: preflight audio ingest, then resolve a stored document.
readiness = client.context.get_audio_readiness(
    workroom_id=my_workroom_id,
    mime_type="audio/aiff",
)
if readiness["ready"]:
    download = client.context.get_document_download_url(
        "urn:source:abc123",
        workroom_id=my_workroom_id,
    )
    url = download["download_url"]
```

Error Handling

```
from kamiwaza_sdk.exceptions import KamiwazaError

# my_workroom_id is a workroom you own. The Global Workroom id is available as
# client.context.DEFAULT_WORKROOM_ID.
try:
    client.context.create_ontology(
        name="my-graph",
        backend="graphiti",
        workroom_id=client.context.DEFAULT_WORKROOM_ID,  # Global → will 403
    )
except KamiwazaError as exc:
    if exc.status_code == 403 and "read-only" in str(exc):
        # Expected: the Global Workroom is a shared, read-only catalog.
        # Retry the write against a workroom you own instead.
        client.context.create_ontology(
            name="my-graph",
            backend="graphiti",
            workroom_id=my_workroom_id,
        )
    else:
        raise
```

Best Practices

1. Treat the Global Workroom as **read-only**: query/search it, but write your own resources into a workroom you own.

2. Pass an explicit `workroom_id` on writes so a resource never accidentally resolves to the Global Workroom (an omitted workroom can resolve to Global on some paths and will be rejected).
3. Branch on `status_code == 403` + `"read-only"` to distinguish the shared-catalog policy from a genuine permission error.
4. Use `query_vectors_global()` / read-only ontology queries to consume shared knowledge; never rely on writing to Global from tenant code.

Embedding Service

Overview

The Embedding Service (`EmbeddingService`) provides comprehensive text embedding functionality for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/embedding.py` , this service handles text chunking, embedding generation, and provider management.

Key Features

- Provider-Based Architecture
- Text Chunking with Metadata
- Embedding Generation
- Batch Processing
- Multiple Provider Support
- Stateless Design

Getting Started

The embedding service requires initializing a provider before use:

```
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient(api_key="your-key")

# Get an embedder instance (required first step)
embedder = client.embedding.get_embedder(
    model="nomic-ai/nomic-embed-text-v1.5",  # optional, this is default
    provider_type="sentencetransformers",     # optional, this is default
    device="cuda"                             # optional, auto-detect if None
)
```

Text Processing

Available Methods

- `chunk_text(text: str, max_length: int = 1024, overlap: int = 102, preamble_text: str = "", return_metadata: bool = False) -> Union[List[str], ChunkResponse]` : Split text into chunks

- `embed_chunks(text_chunks: List[str], batch_size: int = 64) -> List[List[float]]`: Generate embeddings for chunks
- `create_embedding(text: str, max_length: int = 1024, overlap: int = 102, preamble_text: str = "") -> EmbeddingOutput`: Create embedding for text
- `get_embedding(text: str, return_offset: bool = False) -> EmbeddingOutput`: Generate embedding for text

```
# Initialize embedder first
embedder = client.embedding.get_embedder()

# Split text into chunks
chunks = embedder.chunk_text(
    text="Long document text...",
    max_length=1024,
    overlap=102
)

# Or get chunks with metadata
chunk_response = embedder.chunk_text(
    text="Long document text...",
    max_length=1024,
    overlap=102,
    return_metadata=True
)
# Access: chunk_response.chunks, chunk_response.offsets, chunk_response.token_counts

# Generate embeddings for chunks
embeddings = embedder.embed_chunks(chunks, batch_size=64)

# Create single embedding
result = embedder.create_embedding("Sample text")
embedding_vector = result.embedding  # List[float]

# Generate embedding (alternative method)
result = embedder.get_embedding("Sample text")
embedding_vector = result.embedding
```

Provider Management

Getting an Embedder

The primary method for working with embeddings is through `get_embedder()`:

```
# Get embedder with default settings
embedder = client.embedding.get_embedder()

# Get embedder with custom model
embedder = client.embedding.get_embedder(
    model="sentence-transformers/all-mpnet-base-v2",
    provider_type="sentencetransformers",
    device="cuda"  # or "cpu", "mps", None for auto-detect
)

# List available providers
providers = client.embedding.get_providers()
```

Default Configuration

- **Default Model**: `nomic-ai/nomic-embed-text-v1.5`
- **Default Provider**: `sentencetransformers`
- **Default Device**: Auto-detected based on availability

Return Types

The service returns Pydantic models for structured data:

EmbeddingOutput

```
class EmbeddingOutput:
    embedding: List[float]  # The embedding vector
    offset: Optional[int]   # Offset in original text (if requested)
```

ChunkResponse

```
class ChunkResponse:
    chunks: List[str]                    # Text chunks
    offsets: Optional[List[int]]         # Start positions in original text
    token_counts: Optional[List[int]]    # Token count per chunk
    metadata: Optional[List[dict]]       # Additional metadata per chunk
```

Error Handling

The service uses a unified error handling approach:

```python
from kamiwaza_sdk.exceptions import APIError

try:
    embedder = client.embedding.get_embedder()
    result = embedder.create_embedding("text")
except APIError as e:
    print(f"Operation failed: {e}")
```

Deprecated Methods

The following methods are deprecated and should not be used:

HuggingFaceEmbedding()

- **Status**: Deprecated
- **Replacement**: Use `get_embedder()` instead
- **Warning**: Logs deprecation warning when called

reset_model()

- **Status**: Deprecated (no-op in stateless design)
- **Returns**: `{"status": "no-op"}`
- **Note**: Model state is now handled per request

call()

- **Status**: Deprecated
- **Replacement**: Use `get_embedder()` then call methods on the provider
- **Raises**: `DeprecationWarning`

Best Practices

1. **Always initialize an embedder first** using `get_embedder()`
2. **Choose appropriate chunk sizes** based on your model's context window
3. **Use batch processing** for multiple texts to improve performance
4. **Handle overlaps properly** to maintain context between chunks
5. **Consider memory usage** when processing large batches
6. **Cache embeddings** when possible to avoid recomputation
7. **Use return_metadata=True** when you need chunk offsets or token counts

Complete Example

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.exceptions import APIError

# Initialize client
client = KamiwazaClient(api_key="your-key")

# Get embedder
embedder = client.embedding.get_embedder(
    model="nomic-ai/nomic-embed-text-v1.5",
    provider_type="sentencetransformers"
)

try:
    # Process a document
    document = "Your long document text here..."

    # Chunk with metadata
    chunk_response = embedder.chunk_text(
        text=document,
        max_length=512,
        overlap=50,
        return_metadata=True
    )

    # Generate embeddings
    embeddings = embedder.embed_chunks(
        chunk_response.chunks,
        batch_size=32
    )

    # Process results
    for i, (chunk, embedding) in enumerate(zip(chunk_response.chunks, embeddings)):
        print(f"Chunk {i}: {len(embedding)} dimensions")
        if chunk_response.offsets:
            print(f"  Offset: {chunk_response.offsets[i]}")
        if chunk_response.token_counts:
            print(f"  Tokens: {chunk_response.token_counts[i]}")

except APIError as e:
    print(f"Embedding operation failed: {e}")
```

Enclaves Service

Client helpers live in `kamiwaza_sdk/services/enclaves.py` and wrap the `/enclaves/connectors` and `/enclaves/documents` API surfaces.

Usage

```python
from uuid import uuid4

from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.enclaves import ConnectorCreate, IndexDocumentRequest

client = KamiwazaClient("https://localhost/api", api_key="...")

# Connectors
connector = client.enclaves.connectors.create(
    ConnectorCreate(
        name="demo-connector",
        source_type="s3",
        connector_type="s3",
        connection_config={"bucket": "demo"},
    )
)
connectors = client.enclaves.connectors.list(limit=10)
client.enclaves.connectors.trigger_ingest(connector.id)

# Documents
request = IndexDocumentRequest(
    source_id=connector.id,
    source_ref="s3://demo/report.txt",
    item_type="document",
    metadata={"title": "Report"},
)
client.enclaves.documents.create(request)

results = client.enclaves.documents.list(
    source_id=connector.id,
    limit=5,
    marking_level="PUBLIC",
)
print(results.total)
```

> **Routing note:** the FastAPI router is mounted under `/api/enclaves/*`, so the fully-qualified paths are `/enclaves/connectors` and `/enclaves/documents` when using a base URL that already includes `/api`.

Extensions Service

The Extensions Service (`client.extensions`) manages K8s-native extensions deployed as `KamiwazaExtension` Custom Resources. It replaces the legacy App Garden (`client.apps`) and Tool Shed (`client.tools`) services for Kubernetes deployments.

Quick Start

```python
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient(base_url="https://kamiwaza.test/api")
# ... authenticate ...

# List all extensions
extensions = client.extensions.list_extensions()
for ext in extensions:
    print(f"{ext.name}: {ext.phase} ({ext.type})")

# Get a specific extension
detail = client.extensions.get_extension("my-extension")
print(detail.endpoints.external)

# Delete an extension
client.extensions.delete_extension("my-extension")
```

Methods

`list_extensions() -> List[Extension]`

List all extensions visible to the current user.

```python
extensions = client.extensions.list_extensions()
for ext in extensions:
    print(f"{ext.name} ({ext.type}): {ext.phase}")
```

`get_extension(name: str) -> Extension`

Get a single extension by CR name.

```
ext = client.extensions.get_extension("kaizen-abc12345")
print(ext.version)        # "2.0.0"
print(ext.phase)          # "Running"
print(ext.endpoints)      # ExtensionEndpoints(external="https://...",
internal="http://...")
```

Raises: `NotFoundError` if the extension does not exist.

`create_extension(request: CreateExtension, *, wait: bool = False) -> Extension`

Create a new extension from a specification.

Core answers **201** once the deploy is applied, or **202** ("created, not ready") when a direct-runtime deploy must wait for a one-shot Job or a `service_healthy` dependency. Both return an `Extension`; a 202 one has `phase="Provisioning"`, `ready=False`, `is_pending=True`, and `pending_reason` (for example `"waiting for one-shot Job my-tool-migrate to complete"`). Follow it with `wait_for_extension()`:

```
ext = client.extensions.create_extension(request)        # may be Provisioning (202)
ext = client.extensions.wait_for_extension(ext, timeout=600)  # returns when Running
```

`wait=True` sends `?wait=true` so Core blocks and returns the final outcome (including its 503/409 errors) in the create response itself. Cores that predate the 202 answer always block; their result passes through `wait_for_extension()` unchanged.

If Core is already running its maximum number of creates it answers 503 with `Retry-After` before starting the create; the SDK retries that (at most 4 attempts and about 70s of jittered backoff; 3 attempts with Core's 30s hint) and then raises `ExtensionCreateBusyError` (an `APIError`).

TLS verification is configured in each service's environment through `KAMIWAZA_VERIFY_SSL` and `KAMIWAZA_TLS_REJECT_UNAUTHORIZED`. The extension CLI sets both on every service from the selected connection and its TLS override, replacing conflicting Compose values in the generated payload. A full redeploy updates every service. With `kz-ext dev --service X`, only the selected service is updated; run a full redeploy when changing TLS policy for the whole extension. These are Kamiwaza-specific settings; they do not configure arbitrary third-party TLS clients.

For services using the shared Python runtime with a private CA, mount a readable PEM bundle in the container and set `KAMIWAZA_CA_BUNDLE` to its in-container path. Those clients intentionally ignore `SSL_CERT_FILE` and `SSL_CERT_DIR`; trusting the CA on the workstation does not configure container trust. Keep verification enabled in production. For an intentional development-only opt-out, set `KAMIWAZA_VERIFY_SSL=false` in the shell running the deployment; this also disables verification for the CLI's own API calls.

The legacy `KamiwazaIntegrationSpec.tls_reject_unauthorized` attribute remains readable locally but is excluded from serialized requests: the current platform rejects that retired integration field with HTTP 422.

Compose URLs referencing the service itself are scoped to its deployed Service name, just like references from another service. Kubernetes does not provide the unprefixed Compose alias, including for self-callbacks. Scoped Service callbacks can reach any ready replica. Use `localhost` or `127.0.0.1` when a call must stay in-pod, including startup or readiness calls before the pod is ready.

```python
from kamiwaza_sdk.schemas.extensions import CreateExtension, ExtensionServiceSpec

request = CreateExtension(
    name="my-tool",
    type="tool",
    version="1.0.0",
    services=[
        ExtensionServiceSpec(
            name="backend",
            image="myregistry/backend:1.0.0",
            primary=True,
            ports=[{"container_port": 8000}],
        ),
    ],
)
ext = client.extensions.create_extension(request)
print(ext.name)   # "my-tool"
print(ext.phase)  # "Pending"
```

`wait_for_extension(extension, *, timeout=600, poll_interval=2, max_poll_interval=5, on_progress=None) -> Extension`

Block until an extension is ready. `extension` is the `Extension` returned by `create_extension()` or a name. A non-pending `Extension` is returned at once; otherwise `GET /extensions/{name}/status` is polled, backing off from `poll_interval` to `max_poll_interval` seconds, until the phase is `Running` and every service reports `ready_replicas >= replicas` (a completed one-shot Job counts as ready). The result is `GET /extensions/{name}` with `ready=True` (a non-pending input is returned unchanged, so a 201 result keeps `ready=None`; to follow an extension fetched with `get_extension()`, pass its name). `on_progress(line)` is called whenever the observed state changes.

Raises:

- `ExtensionCreateFailed` — the create failed after Core accepted it. Failures are matched by `create_attempt_id` from the create response (Kamiwaza 1.3.0 and later): a `CreateFailed` event or rolled-back 404 with the same id fails the wait at once, and another attempt's record is ignored as stale. Without an id (a name, or an older Core) the record cannot be attributed — Core keeps it per API process, and a replica can still hold a previous same-name attempt's failure — so the event is terminal only with phase `Failed` and is otherwise reported in the `ExtensionWaitTimeout` message. `reason` is Core's explanation: the `CreateFailed` status event (`create_status_code` is the HTTP status the synchronous create would have returned), the rolled-back 404 detail, or the warning events of a `Failed` phase. If the extension disappears (an untagged 404 that persists for 60 seconds, which rides out Core's post-202 read-back race; a 404 tagged with another attempt's id is stale and never ends the wait) and Core does not say why (the failure record is per API process and kept 15 minutes), the error says so; retry with `wait=True` to get the error directly.
- `ExtensionWaitTimeout` — not ready within `timeout`; carries `phase` and `pending_reason`. The deploy is not cancelled.
- Other `APIError`s from polling (for example 403). Up to three consecutive 500/502/503/504 or transport errors are tolerated.

```python
from kamiwaza_sdk.exceptions import ExtensionCreateFailed, ExtensionWaitTimeout

try:
    ext = client.extensions.wait_for_extension(ext, on_progress=print)
except ExtensionCreateFailed as exc:
    print(f"create failed: {exc.reason}")
except ExtensionWaitTimeout as exc:
    print(f"still {exc.phase}: {exc.pending_reason}")
```

`delete_extension(name: str) -> bool`

Delete an extension by CR name. Returns `True` on success.

```python
client.extensions.delete_extension("my-tool")
```

Raises: `NotFoundError` if the extension does not exist.

Note: K8s CR deletion is asynchronous. The CR may still appear briefly in `list_extensions()` after deletion.

Schemas

`Extension`

Response model for an extension.

Field	Type	Description
`name`	`str`	Extension CR name
`type`	`str`	`"app"` or `"tool"`
`version`	`str`	Extension version
`phase`	`str | None`	Phase: `Pending`, `Provisioning`, `Running`, `Failed`, `Unknown`, `Terminating`
`services`	`List[ExtensionServiceStatus]`	Per-service status
`endpoints`	`ExtensionEndpoints | None`	Resolved external/internal URLs
`owner_user_id`	`str | None`	User who created the extension
`created_at`	`datetime | None`	Creation timestamp
`ready`	`bool | None`	`False` on a 202 "created, not ready" create response; `True` from `wait_for_extension()`; otherwise unset
`pending_reason`	`str | None`	On a 202 create response, what the deploy is waiting for
`create_attempt_id`	`str | None`	Kamiwaza 1.3.0 and later: the create attempt behind a 201/202 response, used to match its failure

`CreateExtension`

Request model for creating an extension.

Field	Type	Required	Description
`name`	`str`	Yes	Extension name (K8s DNS label format)
`type`	`"app" | "tool"`	Yes	Extension type
`version`	`str`	Yes	Semver version string
`services`	`List[ExtensionServiceSpec]`	Yes	One or more service specs
`kamiwaza`	`KamiwazaIntegrationSpec`	No	Platform integration settings
`networking`	`NetworkingSpec`	No	Ingress configuration
`security`	`SecuritySpec`	No	Security classification

`ExtensionServiceSpec`

Specification for a single service within an extension.

Field	Type	Default	Description
`name`	`str`	—	Service name
`image`	`str`	—	Container image
`primary`	`bool`	`False`	Whether this is the primary ingress service
`ports`	`List[ExtensionPort]`	`[]`	Exposed ports; a one-shot Job must have none
`depends_on` (`dependsOn`)	`List[str] | None`	`None`	Service names to reconcile first
`depends_on_conditions` (`dependsOnConditions`)	`Dict[str, str] | None`	`None`	Compose conditions for those dependencies
`run_to_completion` (`runToCompletion`)	`bool | None`	`None`	Explicit one-shot Job intent; `None` preserves legacy inference
`env`	`List[Dict]`	`None`	Environment variables
`replicas`	`int`	`1`	Number of replicas
`resources`	`ResourceSpec`	`None`	CPU/memory requests and limits
`command`	`List[str]`	`None`	Container command override

The SDK sends the snake_case field names, which Core publishes and accepts. The camelCase names in parentheses are accepted as input only: older Core builds published them, and the SDK still reads them, but it never sends them. A `service_completed_successfully` dependency fences its consumer until the dependency Job succeeds. On the direct runtime, a create also waits for a `service_healthy` dependency to become healthy before applying its consumer (Core may answer that create 202 while it waits; see `wait_for_extension`). `service_started` only specifies reconciliation order; services that require a started dependency to be ready must poll or retry. `run_to_completion: true` makes a portless, non-primary service with one replica and no runtime dependents a Job. CREATE omits absent dependency and one-shot fields. PATCH can send empty dependency lists/maps to clear old edges; an absent one-shot field preserves the prior value, while an explicit false clears it. Core and the extension operator must support these fields before clients send them.

Migration from App Garden / Tool Shed

Legacy (App Garden / Tool Shed)	Extensions API
`client.apps.list_deployments()`	`client.extensions.list_extensions()`
`client.apps.deploy(template_id=..., name=...)`	`client.extensions.create_extension(request)`
`client.apps.get_deployment(deployment_id)`	`client.extensions.get_extension(name)`
`client.apps.stop_deployment(deployment_id)`	`client.extensions.delete_extension(name)`
`client.tools.list_deployments()`	`client.extensions.list_extensions()`
`client.tools.deploy(template_id=..., name=...)`	`client.extensions.create_extension(request)`

Key differences:

- Extensions are identified by **name** (not UUID)
- Extensions use a **declarative spec** (`CreateExtension`) instead of template IDs
- The extensions API talks to K8s CRDs, not Docker Compose

Related

- Apps Service (deprecated) (see online documentation)
- Tools Service (deprecated) (see online documentation)

Federations Service

Overview

The Federations service (`FederationsAPI` , `kamiwaza_sdk/services/federations.py`) manages cluster-to-cluster federation pairings and cross-cluster access from the SDK. Access it as `client.federations` . It covers the federation lifecycle (pair, list, get, probe, disconnect), a per-federation proxy for sub-resources, and brokered-user allowlisting. Cross-cluster **identity trust** is governed by each federation's identity mode (`peer_kc` or `shared_idp` in Kamiwaza 1.2.0): see the platform Identity Trust Modes (see online documentation) guide for the trust model. The `receiver_realm` value and the SDK's guest helper symbols are reserved for a future receiver-owned identity workflow. The 1.2.0 SDK still accepts `realm_scope` for forward compatibility, but Core 1.2.0 ignores that unknown request field: it does not select `receiver_realm` or return `identity_mode_unsupported` .

Methods

`pair(name, role, *, remote_ips, preshared_key=None, shared_issuer_url=None, ...) -> Federation`

Create a federation pairing. `role` is the side being set up (`initiator` or `receiver`). A pre-shared key is auto-generated (UUID4) when `preshared_key` is `None` ; supply your own to share a key out-of-band.

Selecting the identity mode:

- supplying `shared_issuer_url` (with `shared_ca_pem` when the realm's certificate is from a private CA) creates a **receiver-controlled** **`shared_idp`** federation. The receiver fetches the realm's signing keys from the JWKS URL derived from the issuer; `shared_jwks_url` is accepted but ignored;
- do not supply `realm_scope` on Kamiwaza 1.2.0. Core ignores it, so it is not a mode selector or a fail-closed security signal;
- without `shared_issuer_url` , Core follows the legacy source-trusted **`peer_kc`** path. It creates the federation only when `ALLOW_UNTRUSTED_FEDERATION` permits that path; otherwise it fails with `untrusted_federation_disabled` .

```
# shared_idp (receiver-controlled)
fed = client.federations.pair(
    name="ORION",
    role="receiver",
    # A hostname entry: pairing checks a literal-IP entry against the address
    # the request arrives from, which most ingress paths rewrite.
    remote_ips=[{"ip": "orion.example.com", "primary": True}],
    preshared_key=shared_psk,
    shared_issuer_url="https://idp.example/realms/federated",
)
```

`list() -> List[Federation]`

List all federations on this cluster (`GET /cluster/federations`). Handles both a bare list and an `{"items": [...]}` response shape.

`get(federation_id) -> Federation`

Fetch a single federation by id (`GET /cluster/federations/{id}`).

`by_id(federation_id, *, remote_name=None) -> FederationProxy`

Return a proxy bound directly to an authoritative federation id, without a federation read or list lookup. Use this for receiver-side user and disconnect operations when pairing has replaced an operator-entered label with the peer's advertised cluster name. IDs may be UUID strings or `uuid.UUID` objects; invalid IDs raise `ValueError` before any request is made.

```
receiver = client.federations.by_id(receiver_federation_id)
receiver.users.add(external_id="carol@src-uuid")
```

For a mesh probe, prefer the name-keyed proxy below. If the caller already has both values, `by_id(id, remote_name="ORION")` uses the exact id for control-plane operations and `ORION` as the mesh selector and name-keyed federation-credential lookup. Without `remote_name`, probing uses the id as both the mesh selector and credential key; core accepts a federation UUID selector, and the matching environment-variable suffix is the upper-cased UUID with hyphens replaced by underscores.

`client.federations[name] -> FederationProxy`

Index by name to get a proxy for one federation's sub-resources. The proxy resolves the federation id on first use and caches it.

```
orion = client.federations["ORION"]
caps  = orion.probe()               # peer capabilities over the mesh
orion.users.add(external_id="alice@peer-uuid")
orion.disconnect()
```

`FederationProxy.probe() -> ClusterCapabilities`

Probe the peer's capabilities through the local mesh proxy. Mesh egress is authenticated-only.

`FederationProxy.disconnect(*, force=False) -> Any`

Disconnect (unpair) the federation (`POST /cluster/federations/{id}/disconnect`). `force=True` tears down without waiting for the peer.

`FederationProxy.users.add(external_id, *, initial_tuples=None) -> BrokeredUser`

Allowlist a brokered remote user on this (receiver) cluster (`POST /cluster/federations/{id}/users`). `external_id` identifies the remote subject; `initial_tuples` seeds the ReBAC grants the user should have on this cluster.

Reserved guest helpers

`FederationProxy.guests.enroll(...)` and `.revoke(...)` are forward-looking SDK symbols for the unavailable `receiver_realm` workflow. They are not part of the supported Kamiwaza 1.2.0 server contract and must not be used against a 1.2.0 cluster.

Trust lifecycle (`client.cluster`)

Rotating the pre-shared key, replacing a rotated peer CA, and undoing a disconnect are per-federation-id operations and hang off `client.cluster` (`ClusterAPI`), not the name-keyed `FederationProxy` . All four are admin + native-realm.

`client.cluster.rotate_preshared_key(federation_id) -> dict`

Open a rotation window (`POST /cluster/federations/{id}/rotate-preshared-key`). **Additive** — the outgoing key keeps verifying until the window is closed, so this cannot sever the mesh and takes no acknowledgement. The plaintext key comes back on `preshared_key` and is **not retrievable afterwards**; carry it to the peer's operator out of band. Refuses `rotation_already_in_flight` (409) while a window is open — opening twice would overwrite the outgoing key and strand a peer still signing with the original.

`client.cluster.complete_key_rotation(federation_id, *, acknowledged) -> dict`

Close the window, retiring the outgoing key (`POST /cluster/federations/{id}/complete-key-rotation`). **Subtractive** — a peer still signing with the old key stops working the moment this returns. `acknowledged` must be `True` and is not a check: the cluster cannot observe whether the peer adopted the new key, which is exactly why the operator has to assert it. Refuses `rotation_acknowledgement_required` (400) and `no_rotation_in_flight` (409).

```
fed_id = client.federations.get(...).id
opened = client.cluster.rotate_preshared_key(fed_id)
print(opened["preshared_key"])                  # save now — shown once
# ... deliver it to the peer's operator, and only then:
client.cluster.complete_key_rotation(fed_id, acknowledged=True)
```

`client.cluster.refresh_peer_ca(federation_id, *, ca_pem, acknowledged_fingerprint) -> dict`

Replace the stored peer CA after the peer rotated its own (`POST /cluster/federations/{id}/refresh-peer-ca`). `acknowledged_fingerprint` must match the SHA-256 of the whitespace-normalised `ca_pem` . That is not a security check — you supply both halves — it forces the out-of-band comparison with the peer's operator.

Refuses `peer_ca_required` / `fingerprint_acknowledgement_required` (400) and `fingerprint_acknowledgement_mismatch` (409); the refusal carries the supplied CA's real `fingerprint` so it can be verified out of band.

`client.cluster.reconnect_federation(federation_id) -> dict`

Undo a disconnect this cluster performed, re-admitting the peer's brokered users (`POST /cluster/federations/{id}/reconnect`). Accepts a `DISCONNECTED` federation and nothing else (409 `federation_not_disconnected`) — it reverses a local disconnect, where the realm, key and truststore entry were all preserved; re-pairing is the general flow. Returns the count of users `restored` plus the best-effort Keycloak outcomes.

Reserved source-side credential resolution

The SDK retains forward-looking configuration symbols for a future `receiver_realm` credential. Kamiwaza 1.2.0 does not issue that credential or accept `receiver_realm` federations, so do not configure these symbols for a 1.2.0 deployment. `peer_kc` and `shared_idp` calls continue to use their documented identity paths without this header.

The `kamiwaza-federation` CLI

The SDK ships an operator/test utility, `kamiwaza-federation` (`kamiwaza_sdk.seeding.federation`), that scripts shared_idp stand-up and ReBAC access seeding. Groups: `access` (ReBAC grants), `fed` (pair / status / allow-user / unpair), `dataset` / `gate` / `attr` (gated-retrieval setup), and `idp`.

> **The `idp` group is DEV/TEST-only.** Its `bootstrap` / `persona` subcommands drive the Keycloak **admin** API, which the platform ingress does not expose — they need direct Keycloak access (a port-forward). In production, provision the shared realm through the auth chart's install-time Keycloak init-Job pipeline, not this command.

Secrets are always read from environment variables, never passed on the command line.

Ingestion Service

Client helpers live in `kamiwaza_sdk/services/ingestion.py` and wrap the `/ingestion/ingest` API surface exposed by Kamiwaza. DDE connector triggering uses the separate `/dde/connectors` route.

Usage

```python
from kamiwaza_sdk import KamiwazaClient

client = KamiwazaClient("https://localhost/api", api_key="...")  # or set authenticator

# Active ingestion (runs immediately)
resp = client.ingestion.run_active(
    "s3",
    bucket="kamiwaza-sdk-tests",
    prefix="sdk-integration",
    endpoint_url="http://localhost:19100",
    region="us-east-1",
    aws_access_key_id="minioadmin",
    aws_secret_access_key="minioadmin",
)
print(resp.urns)

# Emit a Metadata Change Proposal
client.ingestion.emit_mcp({"entityType": "dataset", "changeType": "UPSERT",
"entityUrn": resp.urns[0]})

# Register a job for polling (registration does not execute the cron schedule)
from kamiwaza_sdk.schemas.ingestion import IngestJobCreate

job = IngestJobCreate(
    job_id="nightly-s3-sync",
    schedule="0 3 * * *",
    source_type="s3",
    conn_args={"bucket": "...", "recursive": True},
)
client.ingestion.schedule_job(job)
status = client.ingestion.get_job_status(job.job_id)
print(status.status, status.last_run)

# Trigger an approved DDE data-source connector (requires manage permission)
trigger = client.ingestion.trigger_dde_connector_ingest("9e7fc7b1-e817-47a3-b360-
872f09f379bb")
print(trigger.status)
```

Routing note: the FastAPI router is included under `/ingestion`, so the fully-qualified paths are `/ingestion/ingest/run`, `/ingestion/ingest/jobs`, etc.

Live capability verification

`test_run_now_owner_job_and_dde_trigger` requires a private JSON file named by `KAMIWAZA_DDE_TRIGGER_FIXTURE`. It must contain `allow_repeat_trigger: true`, the ID of a dedicated disposable DDE connector, its `expected_source_ref` seed document, and a dedicated `non_manager_api_key`. The non-manager must have read access to that same connector but not manage access. The requester needs platform-admin access to read back the dataset owner grant. Both clients require verified TLS; configure a trusted CA rather than disabling certificate checks. Keep the fixture file outside the repository. The test triggers ingestion and leaves the platform's job and document history in place; do not point it at a shared production connector. Without the approved fixture, the complete-claim test skips rather than emitting a passing capability record.

Federated Jobs Service

`client.jobs` submits local or federated Ray jobs. Existing callers may keep using `run`, `submit_async`, `wait`, and `cancel` without delegated access.

Governed delegated access

Use `DelegatedAccess` to name exact receiver resources and operations. The SDK validates the same closed vocabulary as Core before making a request. Wildcards, duplicates, unknown operations, and an empty access block are rejected.

```
from kamiwaza_sdk import (
    DatasetDelegatedAccess,
    DelegatedAccess,
    ModelDelegatedAccess,
)

job_id = client.jobs.submit_async(
    target_cluster="receiver",
    entrypoint="python summarize.py",
    delegated_access=DelegatedAccess(
        datasets=(
            DatasetDelegatedAccess(
                urn=DATASET_URN,
                operations=("discover", "read", "retrieve"),
            ),
        ),
        models=(
            ModelDelegatedAccess(
                deployment_id=DEPLOYMENT_ID,
                operations=("discover", "chat"),
            ),
        ),
    ),
)
```

Passing a mapping with the same shape remains supported. Omitting `delegated_access` preserves the ordinary job wire format and execution path.

Approved Python dependencies

An isolated delegated job can request exact Python packages from the receiver operator's approved catalog:

```
job_id = client.jobs.submit_async(
    target_cluster="receiver",
    entrypoint="python summarize.py",
    delegated_access=access,
    python_packages=["humanize==4.13.0"],
)
```

Only exact `name==version` coordinates are accepted. URLs, version ranges, extras, markers, duplicate package names, and more than 32 packages are rejected before the request is sent. The receiver resolves each coordinate to its operator-approved SHA-256 hashes and installs it from the configured private Python index. Repository URLs, credentials, and trust material are never part of the SDK request or the Ray job environment. Air-gapped deployments can point the operator-managed repository Secret at an internal PEP 503/691-compatible mirror without changing job code.

The older `pip=` convenience parameter belongs to ordinary Ray runtime environments. Use `python_packages=` for isolated delegated jobs.

In-job runtime client

Managed job code uses `JobRuntimeClient`, which connects only to the private Unix socket created for that job:

```
from kamiwaza_sdk import JobRuntimeClient

with JobRuntimeClient.from_environment() as receiver:
    datasets = receiver.datasets.list_granted()
    models = receiver.models.list_granted()
    rows = receiver.retrieval.collect(dataset_urn=DATASET_URN)
    answer = receiver.models.chat(
        deployment_id=DEPLOYMENT_ID,
        messages=[{"role": "user", "content": summarize(rows)}],
    )
```

Streaming variants are `retrieval.stream` and `models.stream_chat`. Each operation re-reads the platform-owned agent identity and verifies the connected Unix peer before sending a request, so a restarted agent is used safely on the next operation. The client never falls back to a bearer token, password, API key, refresh token, or public HTTP endpoint.

Denied resources raise `DelegatedResourceNotFoundError`; lifecycle revocation raises `DelegationRevokedError`; attestation failure raises `JobIdentityUnavailableError`; grant denial raises `DelegatedOperationDeniedError`; and authority, dependency, or replay-store outages raise `DelegationUnavailableError`.

Lab Service

Overview

The Lab Service (`LabService`) provides comprehensive lab environment management for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/lab.py`, this service handles the creation, management, and deletion of lab environments for development and experimentation.

Key Features

- Lab Environment Management
- Lab Creation and Deletion
- Lab Information Retrieval
- Resource Management

Lab Management

Available Methods

- `list_labs() -> List[Lab]`: List all labs
- `create_lab(lab: CreateLab) -> Lab`: Create new lab
- `get_lab(lab_id: UUID) -> Lab`: Get lab info
- `delete_lab(lab_id: UUID) -> None`: Delete lab

```python
# List all labs
labs = client.lab.list_labs()
for lab in labs:
    print(f"Lab: {lab.name} (ID: {lab.id})")

# Create new lab
lab = client.lab.create_lab(CreateLab(
    name="development-lab",
    description="Development environment",
    resources={
        "cpu": 4,
        "memory": "16Gi",
        "gpu": 1
    }
))

# Get lab details
lab = client.lab.get_lab(lab_id)
print(f"Lab Status: {lab.status}")
print(f"Resources: {lab.resources}")

# Delete lab
client.lab.delete_lab(lab_id)
```

Integration with Other Services

The Lab Service works in conjunction with:

1. Cluster Service
    - For resource allocation
2. Authentication Service
    - For access control
3. Activity Service
    - For tracking lab usage

Error Handling

The service includes built-in error handling for common scenarios:

```python
try:
    lab = client.lab.create_lab(lab_config)
except ResourceError:
    print("Insufficient resources")
except QuotaError:
    print("Lab quota exceeded")
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Clean up unused labs
2. Use descriptive lab names
3. Monitor resource usage
4. Implement proper error handling
5. Set appropriate resource limits
6. Document lab purposes
7. Regular status checks
8. Maintain lab inventory

Performance Considerations

- Resource allocation affects startup time
- Concurrent lab limits
- Resource quotas
- Network bandwidth requirements
- Storage requirements

Lab States

Labs can be in various states:

1. Creating
    - Initial setup
    - Resource allocation
2. Running
    - Fully operational
    - Resources allocated
3. Stopping
    - Cleanup in progress
4. Stopped
    - Resources released
5. Failed
    - Setup or operation failed

Optional markings

`client.markings` exposes the active server profile. Markings are disabled by default; an installed provider does not activate them. The SDK contains no vocabulary or parser and never infers authorization from a label.

This example assumes the active server profile recognizes `Company Private`.

```
config = client.markings.config()
if config.enabled:
    parsed = client.markings.parse("Company Private", default_for="document")
    if parsed.marking is None:
        raise ValueError("The server returned no marking for this input")
    workroom = client.workrooms.create(
        "Project", "persistent", marking=parsed.marking,
    )
    presentation = client.markings.compose([parsed.marking])
```

A marking contains `profile_id`, `profile_revision`, `level_id`, optional `raw_text`, and an `attributes` object validated by the server's selected provider. Resource responses also expose the derived `level_id`. Connector configuration accepts optional `level_boundary` and `default_marking`; document indexing accepts `marking_text` for server normalization. Omitted values carry no SDK default. Partial updates preserve omitted fields; explicit null requests removal, subject to server validation and authorization.

The response display is plain `text`, `background_color`, and `foreground_color`. Render it as text; do not parse it to make access decisions. Use the profile's configured `levels` for selectors, limiting assignments to `assignable` choices.

SDK 1.3 and extension runtime libraries 0.6 must deploy with the corresponding platform update. Identity now carries `marking_level` (Python/Go JSON) or `markingLevel` (TypeScript), from `X-User-Marking-Level`. The header is trusted only after platform ingress authentication; extensions do not mint authority. No provider package is an SDK dependency. Offline deployments must include the selected provider and dependencies with the platform installation media.

Model Service

Overview

The Model Service (`ModelService`) provides comprehensive model management functionality for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/models.py`, this service handles model lifecycle management, including creation, deployment, file management, and configuration.

Key Features

- Model Management (CRUD operations)
- Model File Management
- Model Search and Discovery
- Model Download Management
- Model Configuration Management
- Memory Usage Tracking

Model Management

Basic Operations

```
# Get a specific model
model = client.models.get_model(model_id)

# Create a new model
model = client.models.create_model(CreateModel(
    name="my-model",
    description="My custom model"
))

# List all models
models = client.models.list_models(load_files=True)

# Delete a model
client.models.delete_model(model_id)
```

Model Search

```
# Search for models
models = client.models.search_models(
    query="bert",
    exact=False,
    limit=100,
    hubs_to_search=["huggingface"]
)

# Get model by repo ID
model = client.models.get_model_by_repo_id("bert-base-uncased")
```

Model Download Management

Available Methods

- `initiate_model_download(repo_id: str, quantization: str = 'q6_k') -> Dict[str, Any]`: Start model download
- `check_download_status(repo_id: str) -> List[ModelDownloadStatus]`: Check download progress
- `get_model_files_download_status(repo_model_id: str) -> List[ModelDownloadStatus]`: Get detailed file status

```
# Download a model
download_info = client.models.initiate_model_download(
    repo_id="llama2-7b",
    quantization="q6_k"
)

# Check download status
status = client.models.check_download_status("llama2-7b")
```

Model File Management

Available Methods

- `delete_model_file(model_file_id: UUID) -> dict`: Delete a model file
- `get_model_file(model_file_id: UUID) -> ModelFile`: Get file details
- `get_model_files_by_model_id(model_id: UUID) -> List[ModelFile]`: Get all files for a model
- `list_model_files() -> List[ModelFile]`: List all model files
- `create_model_file(model_file: CreateModelFile) -> ModelFile`: Create new model file

- `search_hub_model_files(search_request: HubModelFileSearch) -> List[ModelFile]`: Search hub files
- `get_model_file_memory_usage(model_file_id: UUID) -> int`: Get file memory usage

```
# List model files
files = client.models.list_model_files()

# Get files for specific model
model_files = client.models.get_model_files_by_model_id(model_id)

# Search hub files
files = client.models.search_hub_model_files(HubModelFileSearch(
    hub="huggingface",
    model="bert-base-uncased"
))
```

Model Configuration Management

Available Methods

- `create_model_config(config: CreateModelConfig) -> ModelConfig`: Create new config
- `get_model_configs(model_id: UUID) -> List[ModelConfig]`: Get all configs
- `get_model_configs_for_model(model_id: UUID, default: bool = False) -> List[ModelConfig]`: Get model configs

```
# Create model configuration
config = client.models.create_model_config(CreateModelConfig(
    model_id=model_id,
    parameters={"temperature": 0.7}
))

# Get configurations for model
configs = client.models.get_model_configs(model_id)
```

Memory Usage Tracking

Available Methods

- `get_model_memory_usage(model_id: UUID) -> int`: Get model memory usage
- `get_model_file_memory_usage(model_file_id: UUID) -> int`: Get file memory usage

```
# Check model memory usage
memory_usage = client.models.get_model_memory_usage(model_id)
```

Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    model = client.models.get_model(model_id)
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Always check model compatibility before downloading
2. Monitor download status for large models
3. Use appropriate quantization for your use case
4. Clean up unused model files to manage storage
5. Keep track of model configurations
6. Monitor memory usage for large models

OpenAI Service

Overview

The OpenAI Service (`OpenAIService`) provides OpenAI API compatibility for Kamiwaza model deployments. This service allows you to use your Kamiwaza-deployed models with the familiar OpenAI Python SDK interface.

Key Features

- OpenAI API compatibility
- Automatic deployment discovery
- Configurable SSL verification
- Support for multiple targeting methods (model name, deployment ID, or direct endpoint)

Usage

Getting an OpenAI Client

The service provides several ways to get an OpenAI client for your deployments:

```
# Initialize Kamiwaza client
client = kz("https://prod.kamiwaza.ai/api/")

# Get client by model name
openai_client = client.openai.get_client(model="Qwen2.5-72B-Instruct-GPTQ-Int4")

# Get client by deployment ID
openai_client = client.openai.get_client(deployment_id="your-deployment-uuid")

# Get client by direct endpoint
openai_client = client.openai.get_client(endpoint="http://your-endpoint:port/v1")
```

Pass exactly one of `model`, `deployment_id`, `repo_id` or `endpoint`. A call with none, or with more than one, raises `ValueError`.

Using the OpenAI Client

Once you have an OpenAI client, you can use it just like you would use the official OpenAI SDK:

```python
# Create a chat completion
response = openai_client.chat.completions.create(
    messages=[
        {"role": "user", "content": "Say hello in Chinese!"}
    ],
    model="model"
)

# Access the response
print(response.choices[0].message.content)
```

Best Practices

1. Use model name targeting when you want automatic failover between multiple deployments of the same model
2. Use deployment ID targeting when you need to ensure you're using a specific deployment
3. Use direct endpoint targeting when you need to bypass deployment discovery
4. Keep the same SSL verification settings as your main Kamiwaza client for consistency

Error Handling

```python
try:
    openai_client = client.openai.get_client(model="my-model")
except ValueError as e:
    print(f"No active deployment found: {e}")
except APIError as e:
    print(f"API error occurred: {e}")
```

Notes

- The OpenAI client is configured with `api_key="local"` since authentication is handled by the Kamiwaza client
- SSL verification settings are inherited from your Kamiwaza client configuration
- The model parameter in OpenAI API calls is ignored since the model is determined by the deployment
- All standard OpenAI SDK features (retries, timeouts, etc.) are availables

Retrieval Service

The `kamiwaza_sdk.services.retrieval.RetrievalService` module drives the job-oriented retrieval API introduced in Kamiwaza 0.7.0.

Creating a job

```
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.retrieval import RetrievalRequest

client = KamiwazaClient("https://localhost/api", api_key="...")

request = RetrievalRequest(
    dataset_urn="urn:li:dataset:(urn:li:dataPlatform:s3,my-bucket/my-key,PROD)",
    transport="inline",
    format_hint="parquet",
    credential_override='{"aws_access_key_id":"...","aws_secret_access_key":"..."}',
)
job = client.retrieval.create_job(request)
if job.inline:
    print("Rows:", job.inline.data)
else:
    print("Transport:", job.transport)
```

Polling status

```
status = client.retrieval.get_job(job.job_id)
print(status.status, status.progress)
```

Streaming output

For SSE jobs (`transport="sse"`), `stream_job` yields raw server-sent-event lines:

```
for event in client.retrieval.stream_job(job.job_id):
    print(event)
```

> **Routing note:** the router is mounted under `/retrieval`, so the live paths are `/retrieval/jobs`, `/retrieval/jobs/{job_id}`, etc.

Flight transport (large datasets)

When the server selects Flight, or when you explicitly request `transport="grpc"` in your `RetrievalRequest`, it returns a gRPC Arrow Flight handshake instead of inline or SSE data. The Flight path requires the optional `flight` extra:

```
pip install "kamiwaza-sdk[flight]>=1.1.0"
```

Consuming Flight batches

```python
import pyarrow as pa
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.retrieval import RetrievalRequest

client = KamiwazaClient("https://kamiwaza.example/api", api_key="...")

# 1. Create a job — the server selects grpc automatically for large data.
request = RetrievalRequest(
    dataset_urn="urn:li:dataset:(urn:li:dataPlatform:s3,my-bucket/big-
file.parquet,PROD)",
    transport="grpc",
)
job = client.retrieval.create_job(request)

# 2. Stream batches over Arrow Flight.
#    Pass ca_cert_path for self-signed or private CA environments.
batches = []
for batch in client.retrieval.flight_batches(job, ca_cert_path="/path/to/ca-
bundle.pem"):
    batches.append(batch)

table = pa.Table.from_batches(batches)
print(table.to_pandas())
```

Production and E2E servers advertise `grpc+tls://` endpoints. TLS is required by default. When the Kamiwaza client was constructed with `verify="/path/to/ca-bundle.pem"` (or `ca_bundle=...`), that path is automatically forwarded to Flight. Empty CA data is rejected. Supplying a CA for a plaintext URI is also rejected because gRPC ignores TLS roots on plaintext transports.

Plaintext is available only as an explicit local-development escape hatch:

```
for batch in client.retrieval.flight_batches(job, allow_insecure=True):
    process(batch)
```

`flight_batches` has a finite one-hour deadline for the complete `DoGet` and stream read, including retry backoffs and endpoint fallback, plus gRPC keepalives. Override the deadline when needed:

```
for batch in client.retrieval.flight_batches(job, timeout_seconds=7200):
    process(batch)
```

A deadline failure raises the typed `FlightTimeoutError` and is never retried.

Mixed-version clusters

Legacy Kamiwaza servers can advertise `"protocol": "kamiwaza.retrieval.v1"` in the Flight handshake rather than `"arrow-flight"`. An omitted discriminator also defaults to that legacy protocol; Flight therefore requires an explicit server advertisement of `"arrow-flight"`. Calling `flight_batches` against such a server raises `TransportNotSupportedError` with a clear message naming the unsupported protocol, instead of producing a confusing connection error:

```
from kamiwaza_sdk.exceptions import TransportNotSupportedError

try:
    for batch in client.retrieval.flight_batches(job):
        ...
except TransportNotSupportedError as exc:
    print(f"Server is too old for Flight: {exc}")
    # Fall back: recreate the job with transport="sse" or transport="inline"
```

Retry, token, and completion semantics

Arrow Flight endpoints are *connection alternatives*, not resume points. The server atomically consumes the single-use handshake token when a `DoGet` is claimed, and does not offer an API for refreshing the handshake per endpoint. The SDK retries an endpoint and falls back only after PyArrow reports a typed `FlightUnavailableError` before any batch has been delivered. It never retries timeouts, server errors, authentication or authorization failures, arbitrary exceptions, or any error after the first batch.

Because the token is consumed when the server claims `DoGet`, retry can only succeed when unavailability occurs before that claim. If an unavailable attempt claimed the token and the next attempt rejects it, the SDK preserves the original typed `FlightUnavailableError` instead of misreporting the secondary rejection as an authentication failure.

If a mid-stream error occurs, the safe recovery path is to restart from job creation:

```
# Recovery pattern after a mid-stream Flight failure
job = client.retrieval.create_job(request)   # fresh job
for batch in client.retrieval.flight_batches(job):
    process(batch)
```

The server reports `total_records=-1`, so the SDK does not invent a row-count integrity check. Instead, after a natural clean stream exhaustion, `flight_batches` reads the job status and requires `COMPLETED`. A clean EOF with any other status—or an unavailable status check—raises `FlightIncompleteStreamError`. If a caller intentionally closes or abandons the iterator early, completion is intentionally not checked.

Serving Service

Overview

The Serving Service (`ServingService`) provides comprehensive model deployment and serving capabilities for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/serving.py` , this service manages Ray cluster operations, model deployment, and inference requests.

Key Features

- Ray Service Management
- Model Deployment
- Model Instance Management
- Model Loading/Unloading
- Health Monitoring
- VRAM Estimation

Ray Service Management

Available Methods

- `start_ray() -> Dict[str, Any]` : Initialize Ray service
- `get_status() -> Dict[str, Any]` : Get Ray cluster status

```
# Start Ray service
status = client.serving.start_ray()

# Check Ray status
ray_status = client.serving.get_status()
```

Model Deployment

Available Methods

- `estimate_model_vram(model_id: UUID) -> int` : Estimate model VRAM requirements
- `deploy_model(model_id=..., repo_id=..., wait=True, timeout_seconds=3600, poll_interval_seconds=5.0, **kwargs) -> Union[UUID, bool]` : Deploy a model. The server accepts the request asynchronously and returns the deployment id immediately; with `wait=True`

(default) the SDK polls client-side until the deployment is ready, with `wait=False` it returns the id right away

- `wait_deployment_ready(deployment_id, timeout_seconds=3600, poll_interval_seconds=5.0) -> UIModelDeployment`: Poll an existing deployment until it reaches `DEPLOYED`; raises `DeploymentFailedError` on a FAILED/ERROR/MUST_REDOWNLOAD terminal status and `TimeoutError` past the deadline
- `list_deployments() -> List[ModelDeployment]`: List all deployments
- `list_active_deployments() -> List[UIModelDeployment]`: List only active deployments with running instances
- `get_deployment(deployment_id: UUID, *, timeout_seconds: float | None = None) -> ModelDeployment`: Get deployment details. Each request is bounded to 30 seconds by default; pass a shorter remaining budget when polling under a deadline.
- `stop_deployment(deployment_id: UUID)`: Stop a deployment
- `get_deployment_status(deployment_id: UUID, *, timeout_seconds: float | None = None) -> str`: Get the deployment status as the platform reports it, e.g. `"DEPLOYED"`, with the same 30-second default per-request bound; pass a shorter remaining budget when polling under a deadline. **Changed:** this used to be declared as returning a `ModelDeployment` and validated the response into one. The route answers with a bare JSON string, so every call raised a pydantic `ValidationError`; no caller can have depended on the old return value. Use `get_deployment()` when the rest of the deployment record is wanted.

```
# Estimate VRAM requirements
vram_needed = client.serving.estimate_model_vram(model_id)

# Deploy a model (blocks until ready via client-side polling)
deployment_id = client.serving.deploy_model(repo_id="org/model-repo")

# Fire-and-forget deploy: get the id immediately, observe readiness later
deployment_id = client.serving.deploy_model(repo_id="org/model-repo", wait=False)
deployment = client.serving.wait_deployment_ready(deployment_id)

# List all deployments
deployments = client.serving.list_deployments()

# List only active deployments (deployed status with running instances)
active_deployments = client.serving.list_active_deployments()
# Each active deployment will have:
# - id: The deployment ID
# - m_id: The model ID
# - m_name: The model name
# - status: The deployment status
# - instances: List of running instances
# - lb_port: The load balancer port
# - endpoint: The HTTP endpoint for the deployment (e.g. https://your-
kamiwaza.example/runtime/models/<deployment-id>/v1)

# Get deployment status — a plain string such as "DEPLOYED"
status = client.serving.get_deployment_status(deployment_id)

# Stop deployment
client.serving.stop_deployment(deployment_id)
```

Model Instance Management

Available Methods

- `list_model_instances() -> List[ModelInstance]`: List all model instances
- `get_model_instance(instance_id: UUID) -> ModelInstance`: Get instance details
- `get_health(deployment_id: UUID) -> Dict[str, Any]`: Get deployment health
- `unload_model(deployment_id: UUID)`: Unload model from memory
- `load_model(deployment_id: UUID)`: Load model into memory

```
# List model instances
instances = client.serving.list_model_instances()

# Get instance details
instance = client.serving.get_model_instance(instance_id)

# Check deployment health
health = client.serving.get_health(deployment_id)

# Load/Unload model
client.serving.unload_model(deployment_id)
client.serving.load_model(deployment_id)
```

Error Handling

The service includes built-in error handling for common scenarios:

```
from kamiwaza_sdk.exceptions import APIError, DeploymentFailedError

try:
    deployment_id = client.serving.deploy_model(repo_id="org/model-repo")
except DeploymentFailedError as e:
    # Terminal FAILED/ERROR/MUST_REDOWNLOAD status observed while waiting
    # for readiness (MUST_REDOWNLOAD = corrupted/incomplete model files).
    # e.deployment_id identifies the in-flight deployment for cleanup.
    print(f"Deployment failed ({e.status}, {e.last_error_code}):
{e.last_error_message}")
except TimeoutError as e:
    # Also carries e.deployment_id so the stuck deployment can be stopped.
    print("Deployment did not become ready in time")
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Always estimate VRAM requirements before deployment
2. Monitor deployment health regularly
3. Use appropriate number of replicas based on load
4. Implement proper error handling
5. Clean up unused deployments
6. Consider using advanced generation parameters for better control
7. Load/unload models to manage memory efficiently

Skills Service

The Skills Service (`client.skills`) wraps the Skills Library endpoints exposed by the Kamiwaza backend. It supports browsing shared skills, importing new draft skills from zip packages, publishing via metadata updates, downloading published packages, and exporting one or more packages for operator workflows.

Quick Start

```python
from kamiwaza_sdk import KamiwazaClient
from kamiwaza_sdk.schemas.skills import SkillLibraryUpdateRequest

client = KamiwazaClient(base_url="https://kamiwaza.test/api")
# ... authenticate ...

# Browse the library
skills = client.skills.list_skills(status="published", page_size=20)
for item in skills.items:
    print(item.name, item.category, item.tags)

# Import a new draft skill package
with open("pdf-generator.zip", "rb") as handle:
    created = client.skills.import_skill_package(
        filename="pdf-generator.zip",
        file_content=handle,
    )

# Publish the imported skill
published = client.skills.update_skill_metadata(
    created.id,
    SkillLibraryUpdateRequest(status="published"),
)

# Download the published package bytes
download = client.skills.download_skill_package(published.id)
print(download.filename, len(download.content))
```

Methods

`list_skills(...) -> SkillLibraryListResponse`

List skills visible to the current caller.

```
skills = client.skills.list_skills(
    q="pdf",
    category="export",
    tag="reporting",
    status="published",
    page=1,
    page_size=50,
)
```

Notes:

- Non-operator callers only see published skills.
- `tag` filters against the top-level tags returned by the backend.

`get_skill(skill_id) -> SkillLibraryDetailResponse`

Fetch detail for a single skill.

```
detail = client.skills.get_skill("5b7f0d58-0dbf-4786-9d7f-8a815bb1a8a8")
print(detail.display_name)
print(detail.package_summary.entries)
```

`import_skill_package(...) -> SkillLibraryDetailResponse`

Create a new draft skill by uploading an Agent Skills-compatible zip package.

```
created = client.skills.import_skill_package(
    filename="chart-generator.zip",
    file_content=package_bytes,
)
assert created.status == "draft"
```

Important:

- Import is the creation path currently supported by the backend.
- This creates a draft skill; it does not publish automatically.

`update_skill_metadata(skill_id, update) -> SkillLibraryDetailResponse`

Update mutable metadata for a skill.

```
updated = client.skills.update_skill_metadata(
    created.id,
    SkillLibraryUpdateRequest(
        status="published",
        metadata={"tags": ["charts", "export"]},
    ),
)
```

Mutable fields currently supported:

- `display_name`
- `category`
- `marking` (optional normalized profile/level/attributes envelope)
- `status`
- `trigger`
- `inputs`
- `metadata`

`download_skill_package(skill_id) -> SkillPackageDownload`

Download the published package for a skill.

```
download = client.skills.download_skill_package(skill_id)
with open(download.filename, "wb") as handle:
    handle.write(download.content)
```

Important:

- This is a read path and follows published-skill visibility rules.
- The SDK preserves `filename`, `content_type`, and raw `content`.

`export_skill_package(skill_id) -> SkillPackageDownload`

Export the current skill package for operator/admin workflows.

```
exported = client.skills.export_skill_package(skill_id)
print(exported.filename)
```

Important:

- Export can be used for draft, published, or archived skills when allowed by backend authorization.

`export_skills_bundle(skill_ids) -> SkillPackageDownload`

Export one or more skill packages as a single zip bundle.

```
bundle = client.skills.export_skills_bundle([first_skill_id, second_skill_id])
print(bundle.filename)  # skills-export.zip
```

`delete_skill(skill_id) -> bool`

Soft-delete a skill.

```
client.skills.delete_skill(skill_id)
```

Returns `True` on success and raises `NotFoundError` if the skill does not exist.

Schemas

Common models live in `kamiwaza_sdk.schemas.skills`:

- `SkillLibraryListItem`
- `SkillLibraryListResponse`
- `SkillLibraryDetailResponse`
- `SkillLibraryUpdateRequest`
- `SkillLibraryExportRequest`
- `SkillPackageDownload`

All response models allow extra fields so the SDK stays forward-compatible with backend additions.

Current markings and portable snapshots (core 1.3)

`export_skill_package` and `export_skills_bundle` return version-2 snapshot ZIPs: `manifest.json` contains current metadata and full marking envelopes; `skills/` contains the unchanged original package ZIPs with SHA256 checksums. The stored source checksum continues to identify the original package. A source download through `download_skill_package` does not include subsequent metadata changes.

Pass a single-entry snapshot directly to `import_skill_package`; its current marking is applied before the first persistence and the result is a draft. Multi-entry snapshots require per-entry imports, not an atomic transaction. Version-1 wrappers are unsupported.

For a raw package, supply `marking=Marking(...)` (or a full envelope dictionary) to apply final protection in the initial multipart request. Omit `marking` to use the package/snapshot value. Explicit `marking=None` sends JSON null, which the server rejects if it would erase existing package protection. Invalid source content or

provider metadata is rejected before writes; an override cannot bypass source validation or conflict with a snapshot's marking. Do not implement initial protection as a separate metadata update after upload.

Tool Shed Service

DEPRECATED: The Tool Shed service (`client.tools`) is deprecated and will be removed in a future release. The Docker Compose-based Tool Shed is being replaced by Kubernetes CRD-based extensions. Use the **Extensions API** (`client.extensions`) instead.

Migration guide:

Legacy (Tool Shed)	New (Extensions API)
`client.tools.list_deployments()`	`client.extensions.list_extensi`
`client.tools.deploy_from_template(template_name=..., ...)`	`client.extensions.create_extens` then `wait_for_extension(...)`
`client.tools.get_deployment(deployment_id)`	`client.extensions.get_extensio`
`client.tools.stop_deployment(deployment_id)`	`client.extensions.delete_extens`

See the Extensions Service documentation (see online documentation) for full details.

The Tool Shed service enables deployment and management of MCP (Model Context Protocol) servers that provide tools and capabilities to AI assistants. These Tool servers can integrate with external services, APIs, and systems to extend AI functionality.

Overview

The Tool Shed allows you to:

- Deploy MCP-compatible Tool servers
- Manage Tool server lifecycle
- Discover available tools and their capabilities
- Monitor Tool server health
- Use pre-built Tool templates for common integrations

Authentication

Important: Tool Shed endpoints require authentication. You must provide valid credentials to access these services.

```python
from kamiwaza_sdk import KamiwazaClient as kz
from kamiwaza_sdk.authentication import UserPasswordAuthenticator

# Create authenticated client
client = kz("http://localhost:7777/api/")
authenticator = UserPasswordAuthenticator(
    username="your_username",
    password="your_password",
    auth_service=client.auth
)
client = kz(
    "http://localhost:7777/api/",
    authenticator=authenticator
)
```

Quick Start

```python
# List available Tool templates
templates = client.tools.list_available_templates()
for template in templates:
    print(f"{template['name']} - {template['description']}")

# Deploy a Tool server from template
tool = client.tools.deploy_from_template(
    template_name="tool-websearch",
    name="my-search-tool",
    env_vars={"TAVILY_API_KEY": "your_api_key"}
)

print(f"Tool deployed at: {tool.url}")
print(f"Status: {tool.status}")
```

Available Methods

Deployment Management

`deploy(image, name, port, env_vars=None)`

Deploy a custom Tool server from a Docker image.

Parameters:

- `image` (str): Docker image for the Tool server
- `name` (str): Name for your deployment
- `port` (int): Port the Tool server listens on

- `env_vars` (dict, optional): Environment variables

Returns: `ToolDeployment` object

`deploy_from_template(template_name, name, env_vars=None)`

Deploy a Tool server from a pre-built template.

Parameters:

- `template_name` (str): Name of the template (e.g., "tool-websearch")
- `name` (str): Name for your deployment instance
- `env_vars` (dict, optional): Environment variables (e.g., API keys)

Returns: `ToolDeployment` object

`list_deployments()`

List all active Tool server deployments.

Returns: List of `ToolDeployment` objects

`get_deployment(deployment_id)`

Get details of a specific Tool deployment.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: `ToolDeployment` object

`stop_deployment(deployment_id)`

Stop and remove a Tool server deployment.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: Success message

Discovery and Health

`discover_servers()`

Discover all Tool servers and their capabilities.

Returns: `ToolDiscovery` object containing:

- `total`: Total number of servers
- `servers`: List of servers with their capabilities

`check_health(deployment_id)`

Check the health status of a Tool server.

Parameters:

- `deployment_id` (UUID): The deployment ID

Returns: `ToolHealthCheck` object with status and protocol information

Template Management

`list_available_templates()`

List all available Tool server templates.

Returns: List of template dictionaries with:

- `name`: Template identifier
- `description`: What the Tool does
- `category`: Tool category
- `capabilities`: List of capabilities
- `required_env_vars`: Required environment variables

Common Use Cases

Deploy a Web Search Tool

```
# Deploy Tavily search Tool
tool = client.tools.deploy_from_template(
    template_name="tool-websearch",
    name="search-assistant",
    env_vars={
        "TAVILY_API_KEY": "your_tavily_api_key"
    }
)

print(f"Search Tool available at: {tool.url}")
```

Deploy a Database Query Tool

```
# Deploy PostgreSQL Tool
tool = client.tools.deploy_from_template(
    template_name="tool-postgres",
    name="db-assistant",
    env_vars={
        "DATABASE_URL": "postgresql://user:pass@host:5432/db"
    }
)
```

Discover Tool Capabilities

```
# Discover all available tools
discovery = client.tools.discover_servers()

print(f"Found {discovery.total} Tool servers:")
for server in discovery.servers:
    print(f"\n{server.name} ({server.status})")
    if server.capabilities:
        print("Capabilities:")
        for cap in server.capabilities:
            print(f"  - {cap.name}: {cap.description}")
```

Monitor Tool Health

```
# Check health of all deployments
deployments = client.tools.list_deployments()

for deployment in deployments:
    try:
        health = client.tools.check_health(deployment.id)
        print(f"{deployment.name}: {health.status}")
    except Exception as e:
        print(f"{deployment.name}: Error - {e}")
```

Tool Templates

Available Tool templates include:

- **tool-websearch**: Web search using Tavily API
- **tool-postgres**: PostgreSQL database queries
- **tool-filesystem**: File system operations
- **tool-github**: GitHub repository interaction

- **tool-slack**: Slack messaging integration

Each template requires specific environment variables (like API keys) which are documented in the template details.

Using Tool URLs

Once deployed, Tool servers provide MCP-compatible endpoints that can be used with:

- AI assistants that support MCP
- Custom integrations
- Direct API calls

The `tool.url` returned after deployment is the public HTTPS endpoint for your Tool server.

Error Handling

```python
from kamiwaza_sdk.exceptions import AuthenticationError, NotFoundError

try:
    tool = client.tools.deploy_from_template(
        template_name="tool-websearch",
        name="my-tool"
    )
except AuthenticationError:
    print("Authentication failed. Check your credentials.")
except NotFoundError:
    print("Template not found.")
except Exception as e:
    print(f"Deployment failed: {e}")
```

Best Practices

1. **Secure Credentials**: Store API keys and secrets in environment variables
2. **Health Monitoring**: Regularly check Tool server health
3. **Capability Discovery**: Use discovery to understand what tools can do
4. **Resource Cleanup**: Stop unused Tool servers to free resources
5. **Template Usage**: Use verified templates for better security

MCP Protocol

Tool servers implement the Model Context Protocol (MCP), which standardizes how AI assistants interact with external tools. The protocol defines:

- Tool discovery and capability reporting
- Input/output schemas
- Authentication methods
- Error handling

See Also

- **Extensions Service (see online documentation)** - The recommended replacement for Tool Shed (K8s-native extensions)
- App Service (see online documentation) - Deploy containerized applications (also deprecated)
- Authentication Guide (see online documentation) - Setting up authentication

VectorDB Service

Overview

The VectorDB Service (`VectorDBService`) provides comprehensive vector database management functionality for the Kamiwaza AI Platform. Located in `kamiwaza_sdk/services/vectordb.py` , this service handles vector storage, retrieval, and similarity search operations.

Key Features

- Vector Database Management
- Vector Storage and Retrieval
- Similarity Search
- Simplified Vector Operations
- Database Lifecycle Management

Vector Database Management

Available Methods

- `create_vectordb(config: CreateVectorDB) -> VectorDB` : Create new vector database
- `get_vectordbs() -> List[VectorDB]` : List all vector databases
- `get_vectordb(vectordb_id: UUID) -> VectorDB` : Get database details
- `remove_vectordb(vectordb_id: UUID)` : Remove vector database

```
# Create vector database
vectordb = client.vectordb.create_vectordb(CreateVectorDB(
    name="my-vectors",
    dimension=768,
    metric="cosine"
))

# List databases
databases = client.vectordb.get_vectordbs()

# Get specific database
db = client.vectordb.get_vectordb(vectordb_id)

# Remove database
client.vectordb.remove_vectordb(vectordb_id)
```

Vector Operations

Available Methods

- `insert_vectors(vectordb_id: UUID, vectors: List[Vector]) -> InsertResponse`: Insert vectors
- `search_vectors(vectordb_id: UUID, query: List[float], k: int = 10) -> List[SearchResult]`: Search vectors
- `insert(vectordb_id: UUID, data: Dict[str, Any]) -> InsertResponse`: Simplified vector insertion
- `search(vectordb_id: UUID, query: str, k: int = 10) -> List[SearchResult]`: Simplified vector search

```
# Insert vectors
response = client.vectordb.insert_vectors(
    vectordb_id=db_id,
    vectors=[
        Vector(id="vec1", vector=[0.1, 0.2, 0.3], metadata={"text": "example"})
    ]
)

# Search vectors
results = client.vectordb.search_vectors(
    vectordb_id=db_id,
    query=[0.1, 0.2, 0.3],
    k=5
)

# Simplified operations
# Insert with automatic vector generation
response = client.vectordb.insert(
    vectordb_id=db_id,
    data={"text": "example text", "metadata": {"source": "doc1"}}
)

# Search with automatic query vector generation
results = client.vectordb.search(
    vectordb_id=db_id,
    query="example query",
    k=5
)
```

Error Handling

The service includes built-in error handling for common scenarios:

```python
try:
    vectordb = client.vectordb.create_vectordb(config)
except DimensionError as e:
    print(f"Invalid dimension: {e}")
except MetricError as e:
    print(f"Invalid metric: {e}")
except APIError as e:
    print(f"Operation failed: {e}")
```

Best Practices

1. Choose appropriate vector dimensions based on your embedding model
2. Select the right similarity metric for your use case
3. Use batch operations for better performance
4. Include relevant metadata with vectors
5. Clean up unused databases
6. Use simplified operations when working with text data
7. Monitor database size and performance
8. Implement proper error handling for vector operations

Kamiwaza Extensions

Welcome to the Kamiwaza Extensions documentation. This section contains detailed documentation for specific Kamiwaza extensions, including apps and tools.

Available Extensions

Select an extension from the sidebar to view its documentation.

If you want to build your own extension, start with the Developer Guide (see online documentation).

Developing Kamiwaza Extensions

This document describes how to build, test, and ship a Kamiwaza extension.

What Is An Extension?

A Kamiwaza extension is a containerized application, service, or tool that runs inside the Kamiwaza platform.

Type	What It Is	Example
App	A multi-service application with a frontend and backend, deployed to the App Garden	AI chatbot, document analyzer, recruitment tool
Service	A backend utility that other extensions can depend on	Vector database, shared cache
Tool	A server that exposes MCP functions and/or a REST API facade via the Tool Shed	Web scraper, code executor, API connector, document processor

Extensions are packaged as Docker images, described by metadata (`kamiwaza.json`), and deployed to Kubernetes via the Kamiwaza platform. Each extension lives in its own repository.

For Core version requirements and selecting compatible releases, see Version Compatibility (see online documentation).

Extension Requirements

Required artifacts:

- `kamiwaza.json` — extension metadata (name, version, description, env defaults)
- `docker-compose.yml` — service definitions
- `Dockerfile` per service — container build instructions
- `README.md` — documentation

Deployment requirements (enforced when deploying to the platform, not during local development):

- No hardcoded host port bindings (`ports: ["8000"]` not `"8000:8000"`)
- No bind mounts (named volumes only)
- Resource limits defined for all services (`deploy.resources.limits`)
- Health endpoint at `GET /health` (apps)
- MCP endpoint at port 8000, path `/mcp` (tools)

Your local `docker-compose.yml` can and should use host port mappings, bind mounts, and whatever else makes local development convenient. When you run `kz-ext dev`, the CLI transforms your compose file to meet deployment requirements automatically — stripping host ports, replacing bind mounts with named volumes, and adding resource limits.

Conventions:

- Image tags follow `{version}` format (no `v` prefix)
- Tool names start with `tool-` or `mcp-`
- Service names start with `service-`

Per-service `healthCheck`

An extension service can declare a per-service health probe in its service spec via a `healthCheck` block. When present, the platform uses it to gate readiness for that service independently of the app-level `/health` endpoint. Useful for tools or services that expose readiness on a non-default path or need custom timing.

```
# inside your service spec
healthCheck:
  path: /healthz
  intervalSeconds: 10   # probe cadence, in seconds
  timeoutSeconds: 3     # per-probe timeout, in seconds
  failureThreshold: 3   # consecutive failures before the service is marked unready
```

All duration fields are **seconds** (not milliseconds). Minimums are `intervalSeconds: 1`, `timeoutSeconds: 1`, `failureThreshold: 1`.

If `healthCheck` is omitted, the platform keeps the previous app-level `/health` probe behavior.

Getting Started

Install the CLI

```
pip install kamiwaza-sdk
```

This installs the `kz-ext` command as part of the Kamiwaza SDK package. Build and deployment logic lives in this package rather than in copied template files in your repo.

Because CLI updates can change how your extension is built and deployed, `kamiwaza.json` declares the CLI version range your extension is compatible with:

```
{
  "kz_ext_version": ">=1.0.0,<2.0.0"
}
```

The CLI checks this on every command and warns (or errors) if there is a mismatch. You can also run a compatibility check explicitly:

```
kz-ext doctor
```

This verifies CLI version compatibility, checks that required tools are installed (Docker, etc.), validates your Kamiwaza connection, and reports any issues.

Optional extras for publishing and conversion:

```
pip install kamiwaza-sdk[publish]   # Adds boto3 for kz-ext publish
pip install kamiwaza-sdk[convert]   # Adds anthropic + openai for kz-ext convert
pip install kamiwaza-sdk[all]       # Both
```

Connect to a Kamiwaza instance

Most extensions need Kamiwaza platform features (model access, auth, workrooms). Set up your connection first:

```
kz-ext login https://your-cluster.kamiwaza.test
```

This authenticates via the Kamiwaza SDK, stores a PAT locally, and configures the CLI to target that instance for both local development and deployment. You only do this once per instance.

Create an extension

```
mkdir my-app
cd my-app
kz-ext create --type app --name my-app
```

This creates a production-ready app with a styled frontend and fully-wired backend:

```
my-app/
├── kamiwaza.json
├── docker-compose.yml
├── frontend/
│   ├── Dockerfile
│   ├── start.mjs
│   ├── package.json
│   ├── tailwind.config.ts
│   └── src/
│       ├── app/
│       │   ├── layout.tsx          # SessionProvider + Google Fonts
│       │   ├── page.tsx            # AuthGuard + dashboard with session/models
│       │   ├── globals.css         # Tailwind + Kamiwaza dark theme
│       │   ├── providers.tsx       # SessionProvider wrapper
│       │   ├── logged-out/         # Logout redirect page
│       │   ├── session/            # Session proxy route
│       │   ├── auth/               # Auth proxy routes
│       │   └── api/[...path]/      # Catch-all backend proxy
│       └── components/
│           └── NavBar.tsx
├── backend/
│   ├── Dockerfile
│   ├── requirements.txt
│   └── app/
│       └── main.py                 # Session router, auth, models, chat
└── README.md
```

No template infrastructure, no `scripts/` directory. The `kz-ext` CLI provides all tooling externally.

The generated code comes pre-wired with the Kamiwaza extension libraries:

- **Backend** (`requirements.txt`): `kamiwaza-extensions-lib` — auth middleware (`require_auth`), model client (`get_model_client`), model discovery (`list_available_models`), session endpoints (`create_session_router`)
- **Frontend** (`package.json`): `@kamiwaza-ai/extensions-lib` — `SessionProvider`, `AuthGuard`, `useSession()`, `useModels()`, proxy utilities (`createProxyHandlers`)
- **Frontend styling**: Tailwind CSS with a Kamiwaza dark theme (custom `kw-*` color tokens, CRT scanline effects, component classes)

These are published packages (`pip install` / `npm install`), not copied source files. The health endpoint, session router, auth dependencies, and model endpoints are already configured in the generated `main.py`. The frontend includes proxy routes for session, auth, and all backend API calls.

The Development Loop

Local development

Your first feedback loop runs containers locally via docker-compose, with the Kamiwaza SDK pointing at your connected instance for platform features:

```
kz-ext dev local
```

This runs `docker-compose up --build` with environment variables configured so that:

- Your backend can call Kamiwaza-managed models via the OpenAI-compatible endpoint
- `KAMIWAZA_USE_AUTH` is set to `false` so the session router returns an anonymous identity
- The SDK can make platform API calls (model discovery, etc.) via your connected instance

You get a working app at `http://localhost:3000` and iterate with normal development tools.

Port auto-detection

If ports are already in use, the CLI automatically remaps to the next available port:

```
$ kz-ext dev local
Port 3000 in use — remapping frontend to 3001
Port 8000 in use — remapping backend to 8001
frontend: http://localhost:3001
backend:  http://localhost:8001
```

This lets you run multiple extensions simultaneously without port conflicts. The internal Docker networking (frontend → backend) is unaffected — only the host-mapped ports change.

Hot reload

Your local `docker-compose.yml` can use host port mappings, bind mounts, and dev server entrypoints — none of the deployment restrictions apply locally. To get hot reload, use the standard mechanisms for your stack:

- **Frontend (Next.js/Vite):** Bind-mount your `src/` directory and use the dev server entrypoint. The generated Dockerfile includes a dev stage for this.
- **Backend (FastAPI):** Bind-mount your `app/` directory and run with `uvicorn --reload`.

These bind mounts and dev entrypoints exist only in your local `docker-compose.yml`. When you run `kz-ext dev` for deployed development, the CLI strips them and uses immutable container images instead.

For extensions that don't need Kamiwaza integration (e.g., a standalone tool that will be accessed by apps but doesn't call Kamiwaza APIs itself), `kz-ext dev local` still works — it just runs docker-compose without the SDK configuration.

Local auth bridge (`--auth`)

By default, `kz-ext dev local` runs with `KAMIWAZA_USE_AUTH=false` so the session router returns an anonymous identity. To exercise the real ForwardAuth flow against your connected instance — useful when iterating on auth-gated routes, role checks, or workroom scoping — pass `--auth`:

```
kz-ext dev local --auth
```

This stands up a local sidecar that injects the same identity headers (`X-User-Id`, `X-User-Email`, `X-Workroom-Id`, etc.) the platform would inject in a deployed extension, sourced from your `kz-ext login` session. `X-Workroom-Id` defaults to the global-workroom sentinel (`ffffffff-ffff-ffff-ffff-ffffffffffff`); set `--workroom <uuid>` to scope to a specific workroom. The bridge also configures Docker `host-gateway` routing so the local backend can reach a host-loopback Kamiwaza ingress (e.g. `https://kamiwaza.test`) without manual `host.docker.internal` plumbing.

`--auth` is most useful for `app` scaffolds that exercise authenticated browser flows. Tool and service scaffolds typically don't need it.

Deployed development

When you need to test in the full platform environment — real ingress, real auth flows, real workroom scoping:

```
kz-ext dev
```

This:

1. Builds Docker images with a unique dev tag (e.g., `1.0.0-dev+abc1234.1711900800`). No version bump needed.
2. Pushes images to the container registry.
3. Creates or updates a `KamiwazaExtension` on the target cluster.
4. Waits for the rollout to complete.
5. Prints the URL.

```
Building my-app (1.0.0-dev+abc1234.1711900800)...
  ✓ backend image built
  ✓ frontend image built
Pushing to your-cluster.kamiwaza.test...
  ✓ extension updated
  ✓ rollout complete (18s)

my-app is running at:
https://your-cluster.kamiwaza.test/runtime/apps/my-app-a1b2c3
```

Iterating

The deployed dev loop is:

1. Change code.
2. Run `kz-ext dev`.
3. Test at the printed URL.
4. Repeat.

Each run builds only what changed (Docker layer caching), pushes a uniquely-tagged image, and updates the running extension. The Kubernetes operator handles the rolling update. No version bumps. No registry builds. No manual redeploy.

> **ROLLOUT CHARACTERISTICS**
>
> The PATCH-update path uses the standard Kubernetes rolling-update strategy on the underlying Deployment. In-flight requests at the moment a pod is replaced may be dropped — under default container settings this typically registers as a small spike of `connection reset` / `502` errors during the rollout window (low-loss, but not strictly zero-downtime). For most dev iteration this is invisible; for live user traffic, design idempotent retries on the client or pin pods with a `lifecycle.preStop` hook in your service spec.

Options:

```
kz-ext dev --no-build         # Push only (image already built)
kz-ext dev --service backend  # Rebuild and update one service only
kz-ext dev --revision custom  # Custom image tag
```

Local SDK development (`--sdk-repo`)

When iterating on the runtime libraries themselves (`kamiwaza-extensions-lib` or `@kamiwaza-ai/extensions-lib`), use `--sdk-repo` to override the published packages with your local SDK source:

```
# Via CLI flag
kz-ext dev local --sdk-repo ~/repos/kamiwaza-sdk
```

This mounts the SDK repo into containers and overrides the published package installs:

- **Python backend**: `pip install -e /sdk/kamiwaza_extensions_lib --no-deps` replaces the PyPI package
- **TypeScript frontend**: `npm pack` + `npm install` from the local `kamiwaza-ai-extensions-lib/` replaces the npm package

The overrides are ephemeral — your extension repo's `docker-compose.yml` and Dockerfiles are never modified.

Repo-local config

Instead of passing `--sdk-repo` every time, create a gitignored config file:

```yaml
# .kz-ext/local.yaml
sdk_repo: /Users/you/repos/kamiwaza-sdk
runtime_libs:
  python: local       # override Python lib (default when sdk_repo is set)
  typescript: local   # override TypeScript lib
build_typescript: true  # auto-build TS package if dist/ is missing or stale
```

Then just run `kz-ext dev local` — it reads the config automatically. CLI flags override the config file.

You can override only one library:

```
sdk_repo: /Users/you/repos/kamiwaza-sdk
runtime_libs:
  python: local
  typescript: published   # keep using npm version
```

Remote deploy with local SDK

The `--sdk-repo` flag also works with `kz-ext dev` for remote deployment:

```
kz-ext dev --sdk-repo ~/repos/kamiwaza-sdk
```

Instead of volume mounts, this bakes the local runtime libraries into the Docker images at build time using Docker BuildKit additional build contexts. The resulting images contain your local lib code and are pushed to the cluster normally.

This requires Docker BuildKit (Docker 20.10+). The CLI checks and fails with a clear message if BuildKit is not available.

Validating the override

`kz-ext doctor` validates the SDK override configuration:

```
$ kz-ext doctor

  ✓ Python 3.11.2
  ✓ Docker 24.0.7
  ✓ Docker Compose v2.23.0
  ✓ Connection: dev-cluster (healthy)

  SDK Override (.kz-ext/local.yaml):
  ✓ SDK repo exists: /Users/you/repos/kamiwaza-sdk
  ✓ Python runtime lib: kamiwaza_extensions_lib/ found
  ✓ TypeScript runtime lib: kamiwaza-ai-extensions-lib/ found
  ✓ TypeScript dist/: built, up to date
  ✓ Template contract: requirements.txt ✓  package.json ✓
```

Authentication

How it works

You do not implement auth protocols. No login pages, no token validation, no OAuth flows, no session storage. The platform handles all of that at the ingress layer before requests reach your code.

What you do is wire up the provided libraries to read user identity from platform-injected headers and, on the frontend, wrap your app in the session/auth components. If you use the generated scaffolding from `kz-ext create`, this wiring is already done for you.

The platform flow:

```
Browser → supported ingress → platform authorization → Your Extension
                                                              ↓
                                                  Request includes identity headers
                                                  set by the platform auth service
```

Your extension receives identity headers on every authenticated request. The exact headers are part of the **platform runtime contract** (see below).

Login redirects

When an unauthenticated user tries to access your extension, the platform handles the redirect automatically:

Direct URL access (user navigates to your extension in the browser):

- The platform detects the unauthenticated browser request and redirects to the login page with a return URL.
- After login, the user is sent back to the original extension URL.
- Your extension is not involved — this happens entirely at the ingress layer.

SPA session expiry (user's session expires while using your app):

- API calls from your frontend will receive 401 responses.
- The `AuthGuard` component handles this automatically — it detects session errors, redirects to login, and restores the session when the user returns.

The scaffolded frontend wraps your app in `SessionProvider` and `AuthGuard` from the start:

```tsx
// This is generated for you by kz-ext create
import { SessionProvider, AuthGuard } from '@kamiwaza-ai/extensions-lib/client';

export default function RootLayout({ children }) {
  return (
    <SessionProvider>
      <AuthGuard>
        {children}
      </AuthGuard>
    </SessionProvider>
  );
}
```

This gives you:

- Automatic redirect to login when the session expires or is missing
- Session state available via `useSession()` hook

- Session expiry countdown tracking
- Configurable public routes that skip auth (e.g., `/logged-out`)
- Loading and redirecting states while auth is being resolved

```
import { useSession } from '@kamiwaza-ai/extensions-lib/client';

function Dashboard() {
  const { session, loading, logout } = useSession();

  if (loading) return <Loading />;

  return (
    <div>
      <h1>Welcome, {session.name}</h1>
      <button onClick={() => logout()}>Logout</button>
    </div>
  );
}
```

You do not need to build a login page, handle OAuth callbacks, manage tokens, or implement session storage. If you use the generated scaffolding, `SessionProvider` + `AuthGuard` + `create_session_router()` are already wired up. If you're integrating into an existing app, you add these yourself — but you still never implement auth protocols.

Reading user identity

Python (FastAPI)

The `kamiwaza-extensions-lib` package extracts identity from the platform's forwarded auth headers:

```python
from kamiwaza_extensions_lib import Identity, get_identity, require_auth,
create_session_router
from fastapi import Depends, Request

# Add session endpoints to your app (/session, /auth/login-url, /auth/logout)
app.include_router(create_session_router())

# Require authentication on a route
@app.get("/api/documents")
async def list_documents(identity: Identity = Depends(require_auth)):
    # identity.user_id, identity.email, identity.name, identity.roles,
identity.workroom_id
    return await get_documents_for_user(identity.user_id)

# Or get identity without requiring auth (returns unauthenticated identity if no
session)
@app.get("/api/public")
async def public_endpoint(request: Request):
    identity = await get_identity(request)
    if identity.is_authenticated:
        return {"greeting": f"Hello, {identity.name}"}
    return {"greeting": "Hello, guest"}

# Require a specific role
from kamiwaza_extensions_lib import require_role

@app.get("/api/admin")
async def admin_only(identity: Identity = Depends(require_role("admin"))):
    return {"admin": identity.email}
```

`create_session_router()` adds three endpoints that the frontend's `SessionProvider` relies on:

- `GET /session` — returns the current user's session info
- `GET /auth/login-url` — builds a login redirect URL
- `POST /auth/logout` — logs out and returns redirect URLs

TypeScript (Next.js)

On the server side, extract identity from request headers:

```typescript
import { extractIdentity } from '@kamiwaza-ai/extensions-lib/server';
import { headers } from 'next/headers';

export default async function Page() {
  const headersList = await headers();
  const identity = extractIdentity(headersList);
  // identity.userId, identity.email, identity.name, identity.resolvedUuid
  return <h1>Welcome, {identity?.name}</h1>;
}
```

In API route handlers, use the proxy utilities to forward requests to your backend with auth headers preserved:

```
import { createProxyHandlers } from '@kamiwaza-ai/extensions-lib/server';

const { GET, POST, PUT, DELETE } = createProxyHandlers({
  target: process.env.BACKEND_URL || 'http://backend:8000',
});

export { GET, POST, PUT, DELETE };
```

Calling Kamiwaza APIs from your backend

When your extension's backend needs to call Kamiwaza platform APIs (e.g., model discovery, data catalog), forward the user's auth headers:

```
from kamiwaza_extensions_lib import KamiwazaExtClient, forward_auth_headers

@app.get("/api/available-models")
async def list_models(request: Request, identity: Identity = Depends(require_auth)):
    client = KamiwazaExtClient.from_env()
    models = await client.get_models(request.headers)  # Forwards auth context
    return models
```

For background tasks that outlive the original request (e.g., long-running document processing), use a service account key:

```
import httpx
import os

# Uses KAMIWAZA_API_KEY (injected by platform) — no user context needed
async def background_task():
    headers = {"Authorization": f"Bearer {os.getenv('KAMIWAZA_API_KEY')}"}
    async with httpx.AsyncClient(verify=False) as client:
        resp = await client.get(f"{os.getenv('KAMIWAZA_API_URL')}/models/",
headers=headers)
        return resp.json()
```

Local development

During `kz-ext dev local`, the CLI injects environment variables (`KAMIWAZA_API_URL`, `KAMIWAZA_USE_AUTH`, etc.) so that the same library code works without a real auth flow. When `KAMIWAZA_USE_AUTH` is `false`, `create_session_router()` returns an anonymous identity and the `AuthGuard` allows access without authentication. Your code doesn't need to change between local and deployed modes.

Workroom context

Extensions are workroom-scoped. The `workroom_id` is available on the `Identity` dataclass:

```
@app.get("/api/projects")
async def list_projects(request: Request, identity: Identity = Depends(require_auth)):
    return await get_projects(workroom_id=identity.workroom_id)
```

You do not need to validate workroom membership — the platform enforces this at the ingress layer before requests reach your extension.

Model Integration

How models work with extensions

Kamiwaza manages model deployments. Your extension declares what kind of model it needs, and the platform provides access. Extensions don't deploy, manage, or health-check models — the platform does that.

Declaring model requirements

In `kamiwaza.json`, declare what your extension needs:

```
{
  "preferred_model_type": "reasoning",
  "env_defaults": {
    "MODEL_CAPABILITIES": "chat,function_calling"
  }
}
```

Using models in your code

The extensions library provides a `get_model_client()` helper that returns an OpenAI-compatible async client:

```
from kamiwaza_extensions_lib import get_model_client, require_auth, Identity
from fastapi import Depends, Request

@app.post("/api/chat")
async def chat(request: Request, messages: list[dict], identity: Identity =
Depends(require_auth)):
    client = await get_model_client(request)
    response = await client.chat.completions.create(
        model="auto",
        messages=messages,
    )
    return response
```

Letting users select a model

For apps that want to offer model choice (e.g., a chatbot where users pick which model to talk to):

```
from kamiwaza_extensions_lib import list_available_models

@app.get("/api/models")
async def available_models(request: Request, identity: Identity =
Depends(require_auth)):
    models = await list_available_models(request)
    return [{"id": m.id, "name": m.name, "type": m.model_type} for m in models]

@app.post("/api/chat")
async def chat(request: Request, model_id: str, messages: list[dict], identity:
Identity = Depends(require_auth)):
    client = await get_model_client(request)
    response = await client.chat.completions.create(
        model=model_id,
        messages=messages,
    )
    return response
```

Streaming

For streaming responses (the common case for chat apps):

```
from starlette.responses import StreamingResponse

@app.post("/api/chat/stream")
async def chat_stream(request: Request, messages: list[dict], identity: Identity =
Depends(require_auth)):
    client = await get_model_client(request)
    stream = await client.chat.completions.create(
        model="auto",
        messages=messages,
        stream=True,
    )

    async def generate():
        async for chunk in stream:
            if chunk.choices[0].delta.content:
                yield f"data: {chunk.model_dump_json()}\n\n"

    return StreamingResponse(generate(), media_type="text/event-stream")
```

Local development

During `kz-ext dev local`, the `KamiwazaExtClient.from_env()` reads `KAMIWAZA_ENDPOINT` (model access) and `KAMIWAZA_API_URL` (API access) from the environment. The CLI configures these to point at your connected Kamiwaza instance. You use real models during local development — no mocks or stubs.

Runtime Environment

When your extension is deployed, the platform injects environment variables and HTTP headers that your code can use.

Platform runtime contract (stable)

These are injected by the extension operator and can be relied on by extensions:

Environment variables:

Variable	Description	Example
`KAMIWAZA_DEPLOYMENT_ID`	Unique identifier for this deployment	`my-app-a1b2c3d4`
`KAMIWAZA_APP_NAME`	Extension name from kamiwaza.json	`my-app`
`KAMIWAZA_APP_PORT`	Port the primary service is exposed on	`8000`
`KAMIWAZA_APP_PATH`	Ingress path prefix	`/runtime/apps/my-app-a1b2c3d4`
`KAMIWAZA_APP_URL`	Full public URL for the extension	`https://cluster.test/runtime/apps/my-app-a1b2c3d4`
`KAMIWAZA_API_URL`	Internal Kamiwaza API endpoint	`http://core-raycluster-headless.kamiwaza.svc:7777/api`
`KAMIWAZA_PUBLIC_API_URL`	Public Kamiwaza API endpoint	`https://cluster.test/api`
`KAMIWAZA_ORIGIN`	Platform origin for CORS	`https://cluster.test`
`KAMIWAZA_USE_AUTH`	Whether auth is enabled	`true`

Identity headers (set by ForwardAuth on authenticated requests):

Header	Description
`X-User-Id`	User's unique identifier
`X-User-Email`	User's email address
`X-User-Name`	User's display name
`X-User-Roles`	Comma-separated list of roles
`X-Workroom-Id`	Workroom the user is operating in
`X-Request-Id`	Unique request identifier

Headers with limited documentation (present in the platform but not yet formally part of the extension contract — use via the libraries rather than reading directly):

Header / Variable	Notes
`X-Auth-Token`	Raw JWT token; prefer forwarding via `forward_auth_headers()`
`KAMIWAZA_API_KEY`	Service account key; injection mechanism TBD

Model variables

Variable	Description	Example
`KAMIWAZA_MODEL_PORT`	Port for the paired model endpoint	`8080`
`KAMIWAZA_MODEL_URL`	Full URL for the paired model	(cluster-internal URL)
`KAMIWAZA_ENDPOINT`	Alias for model endpoint (legacy, may be deprecated)	(cluster-internal URL)

Template variables in kamiwaza.json

In your `kamiwaza.json` `env_defaults`, you can use template variables that are resolved at deployment time:

Variable	Resolves To
`{app_port}`	The assigned port for your extension
`{model_port}`	The assigned port for the paired model
`{deployment_id}`	The deployment identifier
`{app_name}`	The extension name

Example:

```
{
  "env_defaults": {
    "PUBLIC_URL": "https://localhost:{app_port}",
    "CALLBACK_URL": "https://localhost:{app_port}/api/auth/callback"
  }
}
```

Deployment inspection

For deployment inspection and day-to-day operations, prefer the supported SDK and CLI surfaces instead of hardcoding raw platform endpoints in your extension code. In practice that means leaning on commands such as `kz-ext status` and the shared runtime libraries rather than treating the platform's internal API paths as part of your extension contract.

Workroom-scoped runtime sessions

If your extension launches a runtime inside a workroom, use the generated auth and session scaffolding plus the shared SDK helpers instead of wiring token handling by hand.

- Keep `SessionProvider`, `AuthGuard`, and `create_session_router()` in place for browser and session lifecycle behavior.
- Use `KamiwazaExtClient.from_env()` and `forward_auth_headers()` when your backend calls Kamiwaza APIs on behalf of the active user.
- Treat any runtime launch token as opaque and short-lived. Do not parse it, log it, persist it outside the active runtime session, or pass it in a URL query string.
- If a downstream platform call returns `401`, fetch a fresh runtime launch token through the supported platform flow and retry once rather than looping retries.

For the public platform contract around workroom-scoped runtimes, membership enforcement, and collaboration streams, see Workroom Runtime Contract (see online documentation).

Writing portable code

Use the library's config and client classes rather than reading environment variables directly. This ensures your code works in both local and deployed modes:

```python
from kamiwaza_extensions_lib import AuthConfig, KamiwazaExtClient

# AuthConfig reads from environment automatically
config = AuthConfig.from_env()
# config.api_url      — KAMIWAZA_API_URL
# config.use_auth     — KAMIWAZA_USE_AUTH (bool)

# KamiwazaExtClient reads endpoints from environment
client = KamiwazaExtClient.from_env()
# client.api_base     — KAMIWAZA_API_URL
# client.openai_base  — KAMIWAZA_ENDPOINT
```

The CLI configures these environment variables for both `kz-ext dev local` and `kz-ext dev`, so the same code works in both modes.

Debugging

Logs

Stream logs from your deployed extension:

```
kz-ext logs                    # All services
kz-ext logs --service backend  # One service only
kz-ext logs --follow           # Stream continuously
```

Shell access

Exec into a running container:

```
kz-ext shell                    # Primary service
kz-ext shell --service backend  # Specific service
```

Status

Check the deployment state:

```
kz-ext status
```

Shows:

- Extension name, version, and dev revision

- Per-service status (image tag, ready replicas, restart count)
- Ingress URL
- Recent events (image pulls, probe failures, restarts)

Port forwarding

Access a service directly without going through the platform ingress:

```
kz-ext port-forward --service backend --port 8000
# Backend is now available at localhost:8000
```

Useful for testing backend APIs directly, connecting a debugger, or inspecting a database. Auto-detects the service and port if not specified.

Bringing an Existing App to Kamiwaza

If you have an existing containerized application and want to run it as a Kamiwaza extension:

```
kz-ext convert /path/to/existing-app
```

The conversion tool uses an AI agent to analyze and modify your existing code:

1. **Analyze** the existing Dockerfiles, compose files, and application code.
2. **Check requirements** — verify the app meets extension requirements (no host ports, no bind mounts, has health endpoint, etc.) and report what needs to change.
3. **Generate** `kamiwaza.json` — infer metadata from existing configuration.
4. **Modify files** — add `kamiwaza-extensions-lib` to Python backends, `@kamiwaza-ai/extensions-lib` to Node.js frontends, wire in health endpoints, auth middleware, session router, and resource limits.
5. **Report** — show what was created, what was modified, and what needs manual attention.

All file modifications are tracked by git — review with `git diff` and revert anything you disagree with.

The AI agent supports multiple LLM providers, picked in this order:

- **Claude CLI / Codex CLI** (subscription auth): If `claude` or `codex` is on your `PATH` and already authenticated (e.g. you've run `claude login`), `kz-ext convert` reuses that session — no API key needed. This is the most convenient path if you already use either CLI day-to-day.
- **OpenAI-compatible** (API key): Set `OPENAI_API_KEY`. Set `OPENAI_BASE_URL` for Kamiwaza, vLLM, Ollama, or any OpenAI-compatible endpoint.
- **Anthropic** (API key): Set `ANTHROPIC_API_KEY`.
- **No credentials**: Falls back to basic `kamiwaza.json` generation with a compatibility report — useful for a quick read on what would need to change without mutating files.

The agent also handles monorepo layouts (it discovers per-app `Dockerfile` / `package.json` / `requirements.txt` under a configurable subdirectory), and can vendor template files via copy rather than symlink when needed for distroless dev-local builds.

Preview without modifying files:

```
kz-ext convert /path/to/existing-app --dry-run
```

After conversion:

```
cd /path/to/existing-app
git diff                # Review changes
kz-ext validate         # Verify everything passes
kz-ext dev local        # Test locally
kz-ext dev              # Deploy to Kamiwaza
```

Updating Scaffolds When the Template Evolves

When the `kz-ext` CLI ships a newer template (new generated files, refreshed proxy routes, updated `AGENTS.md`, etc.), existing scaffolded extensions can pick up the changes without a full rescaffold:

```
kz-ext update
```

`update` reconciles only **template-owned files** — the wiring scaffolded by `kz-ext create`. Author-owned files (your `src/`, your routes, anything you've added or substantially edited) are never touched. Each template-owned file gets one of three strategies:

- **`overwrite`** — replaced unconditionally; a `.orig` backup is written if your on-disk copy differs from the new template.
- **`preserve_if_modified`** — replaced only if the on-disk copy is identical to the previous template render. If you've edited it, the file is skipped (or, in interactive mode, you see a unified diff and choose `apply` / `keep`).
- **`merge`** — reserved for future smart-merge; v1 behaves as `preserve_if_modified`.

Modes:

```
kz-ext update                    # interactive — prompts on every conflict
kz-ext update --dry-run          # print planned changes only; no writes
kz-ext update --force            # apply all template-owned updates, write .orig on
conflicts
kz-ext update --non-interactive  # fail with non-zero exit on the first conflict (for
CI)
kz-ext update --bootstrap        # adopt current state as baseline for older scaffolds
```

The set of template-owned files is enumerated explicitly in code (not glob-based) — there are no surprises about what `update` can or can't touch. Use `--dry-run` first when in doubt; review the proposed diff with `git diff` after running.

Validation

Before opening a PR:

```
kz-ext validate
```

This checks:

- `kamiwaza.json` is well-formed and complete
- Docker images build successfully
- Compose file is App Garden-compatible
- No host port bindings, no bind mounts
- Resource limits defined
- Health endpoint exists (apps)
- Preview image path is valid (if specified)
- With the unreleased compatibility-aware SDK changes, `kamiwaza_version` uses the supported numeric Core requirement grammar (see online documentation) (if specified)

Testing beyond structural validation is the developer's responsibility. Use whatever test framework fits your stack.

Isolated catalog testing (low-level publisher)

Only the shared extension workflow writes the `dev-info` bucket, and only from successful `develop` builds. Do not use the low-level SDK publisher to write that bucket or a production catalog. For isolated compatibility testing, use a dedicated non-production bucket and image namespace:

```
kz-ext config publish-profile sandbox \
  --registry ghcr.io/my-org/sandbox \
  --catalog-endpoint https://my-account.r2.cloudflarestorage.com \
  --catalog-bucket extensions-dev-sandbox \
  --catalog-credentials aws-profile:my-sandbox-profile

kz-ext publish --stage sandbox --catalog-schema compat-v1
```

This low-level path exercises SDK publication behavior only; it does not validate a Kajiya-owned manifest or authenticated promotion receipt. On a real publish, SDK-built images are resolved to their pushed image digests and written as digest-qualified catalog references. A dry run does not perform that registry resolution or prove the final digest. External or prebuilt images must already carry digest pins. `--revision` supplies the revision tag and catalog revision field; it does not pin an image digest. Canonical release publication requires owner and promotion evidence, and the receipt-backed release route is not yet available.

Replace the example namespace and profile with sandbox-only values. The S3 endpoint may be shared, but the bucket and credentials must be scoped to the sandbox bucket. The sandbox credentials must not be able to write `dev-info` or production buckets. Registry credentials must be limited to the sandbox image namespace and unable to push release images. Prefer the explicit sandbox AWS profile shown here over `env`; when using `env`, every ambient `AWS_*` value and `AWS_PROFILE` must be sandbox-scoped. A `catalog_prefix` is only a key prefix, not an access-control boundary.

Each saved publish profile specifies:

- **registry** — where Docker images are pushed (GHCR, ECR, private registry, etc.)
- **catalog-endpoint** — S3-compatible endpoint for the extension catalog
- **catalog-bucket** — bucket name for the catalog
- **catalog-credentials** — how to authenticate (`aws-profile:<name>` , `env` , etc.)

For isolated CI experiments, the low-level publisher also accepts environment overrides. Each of these five `KZ_PUBLISH_*` values overrides that field from the named profile, including an empty value. A stray value can redirect a publish away from the sandbox. Set all five explicitly and keep them scoped to the sandbox; never set the bucket to `dev-info` or a production catalog:

```
export KZ_PUBLISH_REGISTRY=ghcr.io/my-org/sandbox
export KZ_PUBLISH_CATALOG_ENDPOINT=https://my-account.r2.cloudflarestorage.com
export KZ_PUBLISH_CATALOG_BUCKET=extensions-dev-sandbox
export KZ_PUBLISH_CATALOG_PREFIX=eng-12421
export KZ_PUBLISH_CATALOG_CREDENTIALS=aws-profile:my-sandbox-profile
```

Profiles can be stored at the user level (`~/.kamiwaza/profiles/`) or at the repo level (`.kz-ext/profiles/` — useful for team-shared configs). Repo-level profiles override user-level profiles by name.

Do not commit repo-level profiles that target the shared `dev-info` bucket or production storage. Keep committed examples and shared configs scoped to an isolated sandbox. Registry login credentials used by the publisher must also be unable to push to production/release image namespaces.

You do not need publish profiles for the dev loop — `kz-ext dev` only needs the Kamiwaza instance connection from `kz-ext login` .

Profile JSON shape

Behind the CLI, each profile is just a JSON file at `~/.kamiwaza/profiles/<name>.json` (file mode `0600`). You can edit it directly if you'd rather not re-run the CLI:

```
{
  "name": "sandbox",
  "registry": "ghcr.io/my-org/sandbox",
  "catalog_endpoint": "https://my-account.r2.cloudflarestorage.com",
  "catalog_bucket": "extensions-dev-sandbox",
  "catalog_credentials": "aws-profile:my-sandbox-profile",
  "catalog_prefix": "",
  "created_at": "2026-05-06T00:00:00.000000+00:00"
}
```

The `catalog_credentials` field accepts:

- **`aws-profile:<name>`** — `kz-ext` runs `boto3.Session(profile_name="<name>")`, which reads `~/.aws/credentials`. This is the most reliable option for Cloudflare R2 (where you've already configured an `r2.cloudflarestorage.com` profile in `~/.aws/config` with `region = auto`).
- **`env`** — `kz-ext` runs `boto3.Session()` with no args. boto3 then resolves credentials from the standard chain: `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY` exported in the shell, or `AWS_PROFILE` if set. **Note:** `kz-ext publish` does not load `.env` files — these env vars must be exported in the shell that runs the command (or set in your CI environment). If you have a repo-level `.env` with custom variable names like `AWS_PROFILE_DEV=…`, those are not picked up; either export them with `eval $(grep '^AWS_' .env | xargs)` first, or switch the profile to `aws-profile:<name>` (recommended).
- **`sso`** — reserved for Cloudflare SSO; not yet implemented for `kz-ext publish`. Use `aws-profile:<name>` with an SSO-configured AWS profile instead.

Endpoint and credentials must match

The `catalog_endpoint` and the credentials you point at have to be for the same provider. R2 keys talk to R2 endpoints; AWS S3 keys talk to AWS S3 endpoints. Common signs of a mismatch:

- `Unable to locate credentials` — boto3 can't find any credentials at all. Either `~/.aws/credentials` has no `[<profile>]` section by that name, or `catalog_credentials` is `"env"` and no AWS env vars are exported. Fix: confirm `aws --profile <name> sts get-caller-identity` works from your shell, then make sure your kz-ext profile points at that AWS profile.
- `An error occurred (InvalidAccessKeyId) when calling the GetObject operation` — boto3 found credentials, but the *provider* rejected them. This usually means the endpoint and the keys are for different providers (e.g., R2 keys hitting `s3.amazonaws.com`, or AWS keys hitting `r2.cloudflarestorage.com`). Fix: align `catalog_endpoint` with whichever provider holds the keys.

A quick way to check whether an AWS profile is configured for R2: `grep -A2 '\[profile <name>\]' ~/.aws/config` — R2 profiles typically have `region = auto`.

Preparing for a future release

The 1.3.0 canonical release catalog route is not available yet. The shared extension workflow publishes development discovery from successful `develop` builds but does not publish canonical release catalogs. The SDK's direct `kz-ext publish` command does not validate Kajiya-owned manifests or authenticated promotion receipts, so it is not a release command. Publication must wait for the Kajiya receipt-backed route.

When your extension is ready for a wider audience, you can prepare its manifest and validate it. Do not use a sandbox publication as the canonical release or reserve its immutable version for production. Until the receipt-backed release route exists, a version bump is optional:

1. Optional: bump the version in `kamiwaza.json` when preparing a sandbox-only compatibility experiment:

```
{
  "version": "1.1.0"
}
```

2. Validate the extension. This prepares it for a future release; it does not publish it:

```
kz-ext validate
```

Do not use `kz-ext publish` with production credentials. `--force` does not authorize or repair a release.

CI/CD

`kz-ext` works in automated pipelines. Use PAT-based authentication (no interactive login):

```
# In your CI environment
export KAMIWAZA_API_KEY=pat-...
export KAMIWAZA_URL=https://your-cluster.kamiwaza.test/api

# Validate
kz-ext validate
```

The shared workflow keeps development discovery separate from release publication. A successful development catalog update is not a promotion receipt and does not qualify a release.

Extension Anatomy

kamiwaza.json

Every extension has a `kamiwaza.json` at its root. This is the single source of truth for the extension's identity and version.

```
{
  "name": "my-app",
  "version": "1.0.0",
  "source_type": "kamiwaza",
  "description": "A brief description of what this extension does",
  "risk_tier": 1,
  "verified": false,
  "env_defaults": {
    "PUBLIC_URL": "https://localhost:{app_port}",
    "CALLBACK_URL": "https://localhost:{app_port}/api/auth/callback"
  },
  "preferred_model_type": "reasoning",
  "preview_image": "images/preview.png",
  "kamiwaza_version": ">=0.8.0"
}
```

docker-compose.yml

Your development compose file. This is what you use for `kz-ext dev local` and what the tooling transforms for App Garden deployment.

```
services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - BACKEND_URL=http://backend:8000

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - backend_data:/app/data
    environment:
      - OPENAI_BASE_URL=${KAMIWAZA_ENDPOINT:-http://host.docker.internal:8080}
      - OPENAI_API_KEY=${OPENAI_API_KEY:-not-needed-kamiwaza}

volumes:
  backend_data:
```

When deploying to Kamiwaza, the CLI automatically transforms this for the cluster: host ports removed, bind mounts removed, build contexts replaced with image references, tags synced from `kamiwaza.json`. References to `host.docker.internal` (used for local dev to reach the host machine) are replaced by cluster-internal service URLs that the platform injects automatically via environment variables.

Key Patterns

Health endpoint

Apps must expose a health check:

```
@app.get("/health")
async def health_check():
    return {"status": "healthy"}
```

Frontend-backend communication

In Docker, services communicate by service name:

```
const BACKEND_URL = process.env.BACKEND_URL || 'http://backend:8000';

export async function GET(request: NextRequest) {
  const response = await fetch(`${BACKEND_URL}${request.nextUrl.pathname}`);
  return NextResponse.json(await response.json());
}
```

MCP tools

Tools implement the MCP protocol on port 8000:

```
from mcp import FastMCP

mcp = FastMCP("my-tool")

@mcp.tool()
async def search(query: str) -> dict:
    """Search for documents matching the query."""
    results = await do_search(query)
    return {"results": results}

app = mcp.create_fastapi_app()
```

Command Reference

Command	What It Does
`kz-ext login <url>`	Connect to a Kamiwaza instance
`kz-ext create --type app --name my-app`	Create a new extension
`kz-ext dev local`	Run locally with docker-compose + SDK (auto-detects port conflicts)
`kz-ext dev local --auth`	Run locally with the ForwardAuth identity bridge enabled
`kz-ext dev local --sdk-repo <path>`	Run locally with local SDK runtime libs (volume mount)
`kz-ext dev`	Build, push, and deploy to Kamiwaza
`kz-ext dev --sdk-repo <path>`	Build with local SDK libs baked into images, then deploy
`kz-ext dev --no-build`	Push and deploy (skip build)
`kz-ext dev --service backend`	Rebuild one service only
`kz-ext update`	Reconcile a scaffold against the current template (interactive by default; supports `--dry-run` / `--force` / `--non-interactive` / `--bootstrap`)
`kz-ext bump <major|minor|patch>`	Bump the version in `kamiwaza.json`
`kz-ext validate`	Check extension requirements
`kz-ext doctor`	Check CLI compatibility, tools, connection, SDK override config, and per-failure-class auth hints
`kz-ext convert /path`	AI-powered conversion of existing app to extension
`kz-ext config publish-profile <name>`	Configure a named publish profile
`kz-ext config publish-profile --list`	List all publish profiles
`kz-ext publish --stage <profile> --catalog-schema compat-v1`	Low-level publish to an isolated compat-v1 catalog; without this option the schema defaults to legacy v3
`kz-ext publish --stage <profile> --catalog-schema compat-v1 --dry-run`	Preview; does not resolve live registry digests
`kz-ext publish --stage <profile> --catalog-schema compat-v1 --revision <sha>`	Set revision tag and catalog revision; it does not provide a digest pin

Command	What It Does
`kz-ext status`	Show deployment status and URL
`kz-ext logs [--service name] [--follow]`	Stream logs from deployed extension
`kz-ext shell [--service name]`	Exec into a running container
`kz-ext port-forward [--service name] [--port N]`	Forward a port locally

The low-level publisher is not a canonical release route. Do not use it with production credentials, point it at the shared `dev-info` bucket, or treat a catalog write as a Kajiya promotion receipt. Canonical releases require authenticated promotion evidence.

FAQ

Do I need to bump the version to test a change?

No. During development, `kz-ext dev` generates a unique image tag on every build using the git SHA and a timestamp. Version bumps are only needed when you release.

Can I develop without a Kamiwaza instance?

Partially. `kz-ext dev local` runs your containers with plain docker-compose, but most extensions need Kamiwaza features (model access, auth). Connect to a Kamiwaza instance for the best local development experience.

How do I iterate on the runtime libraries?

Use `kz-ext dev local --sdk-repo ~/repos/kamiwaza-sdk` or create a `.kz-ext/local.yaml` config. This overrides the published packages with your local SDK source — no compose surgery needed. See the Local SDK development (see online documentation) section.

How do I update my tooling?

```
pip install --upgrade kamiwaza-sdk                    # CLI + SDK
pip install --upgrade kamiwaza-extensions-lib          # Python runtime library
npm update @kamiwaza-ai/extensions-lib                 # TypeScript runtime library
```

After upgrading, run `kz-ext doctor` to verify compatibility with your extension's declared `kz_ext_version` range. The intelligence lives in installed packages, not copied files in your repo — but a small set of generated wiring (proxy routes, `AGENTS.md`, etc.) is template-owned. To pick up template changes shipped with a newer CLI, run `kz-ext update` (see Updating Scaffolds When the Template Evolves (see online documentation)).

How do I handle authentication?

You don't. The platform authenticates requests before they reach your extension and injects identity headers. Use the SDK middleware to read user identity. See the Authentication (see online documentation) section.

How do I access models?

Use the `get_model_client()` helper from `kamiwaza-extensions-lib`. It returns an OpenAI-compatible async client. See the Model Integration (see online documentation) section.

What is the difference between `kz-ext dev` and `kz-ext publish`?

`kz-ext dev` is the fast inner loop — no registry build or catalog artifacts, unique development tags, and immediate deployment. `kz-ext publish` is a low-level SDK command for an isolated non-production catalog. The shared workflow alone writes development discovery. Neither command authorizes a canonical release. Release publication requires Kajiya's authenticated promotion-receipt path, which is not available yet.

How do I convert an existing app?

Run `kz-ext convert /path/to/app`. The AI agent analyzes your existing code, generates `kamiwaza.json`, and wires in SDK integration. All changes are git-tracked. See the Bringing an Existing App to Kamiwaza (see online documentation) section.

Does `kz-ext` work in CI?

Yes. Use `KAMIWAZA_API_KEY` and `KAMIWAZA_URL` environment variables for headless authentication. See the CI/CD (see online documentation) section.

Compatible Extension Catalogs

> **⚠ UNRELEASED — CANONICAL RELEASE PUBLISHING IS NOT ENABLED**
>
> This page describes coordinated extension-selection changes to Core, the SDK publisher, and Helm catalog packaging. It does not announce availability in a released Core build. The receipt-backed canonical release route is not currently available; there is no supported manual production catalog write or production enablement step. Isolated SDK publisher tests on a dedicated non-production sandbox remain available.

Availability and selection

An installation keeps three distinct records:

Record	Meaning	What changes it
Shipped baseline	Exact extension releases packaged with this Core release	A separately identified Core release baseline
Available releases	Retained published releases and their compatibility/status	Catalog refresh or explicit offline import
Local selection	Release used for subsequent deployments	Explicit administrator action

A connected installation can discover compatible extension releases published after its Core release, including entirely new extensions. It does not need a new Core release for each discovery. **Check for updates** refreshes availability; it does not change selections or deployments. An unavailable online source does not erase locally retained records or replace a selected release.

The shipped baseline remains an immutable reference. Baseline membership and release provenance must be distinguished from a publisher's declaration of compatibility. Declaring a version range does not certify every combination in that range.

Administrator selection controls

The catalog shows the selected release and latest compatible release separately, for example: `DDE Tool | Selected: 1.1.0 | Available: 1.1.2`.

- **Update** selects the displayed compatible target for new deployments.
- **Choose version** selects a retained compatible release, including the shipped baseline or a later compatible release.
- **Update all** previews the exact changes to existing selections and requires confirmation. It does not add newly discovered extensions. A stale or invalid proposal must be reviewed again; confirmation cannot silently choose a different release.

- Newly discovered extensions appear separately with **Add**.
- Extensions outside the shipped baseline support **Remove selection**. This removes their selection for new deployments and preserves existing deployments.

Selection changes require administrator privileges, enforced by both the API and UI. Other authorized users can inspect availability. Release notes are omitted until the catalog provides a supported target.

Applies to new deployments. Existing deployments retain their effective artifacts and settings, including when they restart. Selection rollback does not roll back running services, their databases, or their data. It does not require deleting templates or retiring existing workloads.

The legacy image-only **Upgrade All** operation does not support catalog-managed deployment snapshots. It rejects those deployments instead of creating a mismatch between the running workload and its restart configuration. Use a separately planned deployment replacement when changing running services.

A missing, withdrawn, or incompatible selected release remains visible with its status. A release omitted from an online listing can still be explicitly chosen from retained immutable bytes; it is labeled unlisted and is not proposed by Update all. Explicit withdrawal is a separate blocked state. Core must not substitute a different release. Resolve the condition or explicitly choose a permitted release before creating a new deployment. Withdrawal remains blocked even if a later listing omits or relists the release; there is no automatic reinstatement. Changing the online source retires the old source's update candidates while retaining their bytes for explicit selection. Invalid or conflicting online rows are quarantined with a visible warning so they do not prevent discovery of valid new releases. Explicit imports remain atomic.

Selection changes preserve the installation's risk floor and resource sizing, but invalidate approval of the previous artifact. An administrator must review the newly selected artifact where local governance requires approval. Managed catalog releases cannot be replaced through developer overlays; use a separate custom template name for local builds.

Offline availability imports

For disconnected Helm installations set `core.templates.availability.enabled=false`. The shipped baseline and retained selections remain available. An administrator can import an exported catalog with `POST /api/apps/catalog/import`: supply `releases` as the array of full immutable release objects, `template_types` as an optional map from extension names to `app`, `service`, or `tool`, and `source` as a provenance label such as `offline-import`. Supply the type map for unprefixed legacy tool names. The import is atomic and refreshes availability only; use **Add** or **Choose version** explicitly afterward. A catalog import does not transfer container images: those exact artifacts must also be reachable through the installation's offline registry arrangement.

Declare a Core requirement

Use `kamiwaza_version` in `kamiwaza.json`, independently of the extension's own `version`. These fields describe extension 1.1.2 requiring Core 1.3.1 or newer:

```
{
  "name": "example-extension",
  "version": "1.1.2",
  "kamiwaza_version": ">=1.3.1"
}
```

This is a metadata excerpt, not a complete manifest. `kz_ext_version` separately constrains the SDK CLI.

Requirements accept numeric `major.minor[.patch]` versions with `>=`, `>`, `<=`, `<`, `==`, `!=`, bare equality, and comma-separated AND clauses, such as `>=1.3.0,<1.4.0`. Missing, null, empty, and whole `*` requirements preserve legacy unrestricted semantics; they are not certification evidence. Malformed clauses, nonstring values, `~=`, and partial wildcards such as `==1.*` are rejected.

Extension releases use semantic version precedence. New publication uses complete `major.minor.patch` versions; prerelease identifiers order according to SemVer, and build metadata does not increase precedence. Legacy imported choices retain their original content and provenance rather than being silently rewritten. Automatic update proposals and development baseline projection exclude prereleases. Administrators can explicitly choose compatible retained prereleases. If distinct build identities tie for latest precedence, choose an exact release explicitly.

For example:

Extension release	Core requirement	Discoverable on
1.1.0	`>=1.3.0`	1.3.0, 1.3.1, 1.4.0
1.1.2	`>=1.3.1`	1.3.1, 1.4.0
1.2.0	`>=1.4.0`	1.4.0

An installation selecting 1.1.0 keeps that selection when 1.1.2 or 1.2.0 becomes available. Its administrator decides whether to update. These are compatibility examples, not results from qualification of released builds.

Immutable publication

The SDK contains a history-preserving `compat-v1` publisher with conditional object writes, immutable release identities, and idempotent retries. Those properties describe the publisher implementation; they do not authenticate a release or establish that an extension passed qualification.

For the 1.3.0 release flow, the shared extension workflow writes development discovery to `dev-info` only from successful `develop` builds, using the existing numeric v2/v3 catalog formats. It does not publish canonical release catalogs. Do not point a local `kz-ext publish` profile at `dev-info`. The canonical release mirror is reserved for a Kajiya-selected projection of committed owner manifests and authenticated successful-promotion receipts. That receipt-backed path is not yet available, so there is currently no supported command for publishing a canonical release catalog.

Do not run `kz-ext publish` with production credentials, including `--catalog-schema compat-v1` or legacy numeric catalog schemas. The direct SDK publisher does not validate a Kajiya owner manifest or promotion receipt; a successful catalog write is not evidence of a successful promotion. Keep publisher tests and isolated compatibility experiments on non-production storage.

In isolated SDK publisher use, the history-preserving generation writes `garden/compat-v1/apps.json` for apps and services, and `garden/compat-v1/tools.json` for tools. Publication retains earlier versions, including maintenance releases, and preserves compatibility metadata. Conditional object writes protect concurrent isolated publishers. This is SDK publisher behavior on non-production storage; it does not enable multiple writers for the canonical release mirror.

A `compat-v1` release identity and its payload are immutable in the SDK publisher. Republishing identical content is idempotent. Changing artifacts, compatibility requirements, or other release metadata requires a new version; `--force` cannot overwrite an existing `compat-v1` entry. Runtime image references must be digest-pinned, including external/prebuilt services and declared supporting images. A pinned catalog entry with a mutable image tag does not satisfy artifact immutability. Registry retention must keep referenced artifacts available; Core never substitutes a newer image when an exact artifact is missing.

The shared extension workflow's development route and the future canonical release mirror are separate contracts. Local or manual legacy v2/v3 writes are not release authority and must not be used to publish a release catalog. Existing Core releases do not read `compat-v1` catalogs; that reader requires the coordinated Core update.

Shipped catalog and offline use

Helm supplies the packaged baseline independently of the connected availability source. Online catalogs and operator-imported catalogs do not redefine which releases shipped with Core. Snapshot generation preserves multiversion availability separately from the exact baseline projection.

Offline installations use their shipped baseline and explicitly imported available releases. Importing availability does not silently select releases. Administrators use the same explicit selection controls afterward. Restart and repeated Helm initialization preserve selections, including removed selections. Combined offline imports must preserve both the `releases` array and the `template_types` mapping from `available-releases.json`; the mapping identifies apps, services, and tools without rewriting immutable release payloads.

An absent, invalid, or mismatched shipped baseline produces a visible initialization error. Core preserves existing selections and deployments, and refuses selection changes until the matching immutable baseline is repaired. A fresh installation waiting for its baseline initializes once after repair; later repairs cannot resurrect an explicitly removed selection.

Chart publication requires `--expected-core-version` independently of the chart candidate version and verifies the frozen baseline before running Helm. Active Kajiya chart preparation obtains that expected version from the locked Core Git blob and records its hash. Qualification must still check the exact installed image's runtime version, particularly when a nightly image producer predates the locked source. Packaging never relabels or regenerates the shipped set silently.

Migration captures existing imported choices before any availability refresh can change them. It preserves local/developer templates separately. Legacy mutable artifacts retain legacy provenance; migration does not retrospectively prove their immutability or reconstruct lost historical deployment content.

For a private test source, configure the availability endpoint explicitly: `core.templates.availability.url` in Helm sets `KAMIWAZA_EXTENSION_AVAILABILITY_URL`, which **Check for updates** uses independently of legacy LOCAL catalog settings. The chart requires a trusted HTTPS catalog root. Setting only the legacy LOCAL URL does not redirect connected availability refresh. For offline qualification, disable `core.templates.availability.enabled` and explicitly import local releases. Core workers can read a trusted `file:///absolute/catalog/root` through the legacy LOCAL path. Keep production catalogs untouched when qualifying a new publisher or reader.

The active installation path is Helm/KKS. These selection controls do not require modifying legacy Kajiya bundle-import behavior or replacing canonical owner manifests with another writable inventory.

Verify behavior

Qualification should demonstrate discovery without selection changes, explicit administrator updates, an exact update-all preview, selection rollback, persistence across restart, offline import, and preserved migrated choices. Inspect actual artifact digests of both new and existing deployments, including existing workloads after restart/reconciliation. Include nonadministrator denial and conflicting release-publication tests.

When testing with straw Core identities such as 1.3.0, 1.3.1, and 1.4.0, report those as simulated compatibility-version tests on the actual installed build. They are not proof that those released Core builds were installed or certified.

Historical extension selection qualification snapshot

This page records implementation and isolated-instance evidence collected for the catalog-selection changes. The linked `receipt.json` is historical qualification evidence, not a Kajiya promotion receipt; this page does not announce that the receipt-backed 1.3.0 release catalog route is enabled. For the current publishing boundary, see Immutable publication (see online documentation).

The evidence package is in the Kamiwaza repository at commit `3604f3616c7cb74f03ffdcc9d9449ed211a80eaa`, committed 2026-09-16 15:02 UTC. It records a full matrix against Core source commit `f19983b8a778015b9547f76820b75c4e3a15b2a7` (committed 2026-09-16 14:20 UTC) and a later runtime checkpoint at `82a81921f8a4152c0f404ad97f51fbf274079229` (committed 2026-09-16 14:50 UTC). All cases use simulated Core identities and do not qualify released 1.3.0 binaries.

The implementation snapshot summarized here was exercised on an isolated full Kamiwaza installation with k0s/KKS, PostgreSQL, Keycloak, SpiceDB, the normal API/frontend, and task-owned Moto catalog storage. No production Cloudflare catalogs were written.

The installed candidate uses simulated canonical Core identities 1.3.0, 1.3.1, and 1.4.0. These are actual instance/API/browser/workload tests, **not** tests of released binaries bearing those versions or certification of those combinations.

Automated regression coverage

- Combined Garden, semantic-version, database, Istio routing, setup/audit and route-auth coverage: **4,316 passed, 8 skipped**.
- SDK extension suite: **2,303 passed, one BSD-sed platform skip**.
- Shared-workflow contracts: **233 passed**. Active Kajiya workflow contracts: **38 passed**, plus 8 baseline identity tests. Deploy baseline/publication tests: **147 passed**, plus **20 tenant namespace contract tests**.
- Focused frontend deployment/selection suites: 106 passed before final review; final garden component suite: 18 passed. Production webpack build passed.
- Shared SemVer corpus covers numeric ordering, prereleases, build identities, malformed identities, and legacy normalization consistently across Core/SDK/Deploy.
- Canonical migration `20260916_016` adds durable catalog state and widens template version storage. SQLite initialization is separately exercised.

Working-instance cases

Simulated Core	Shipped fixture	Latest compatible fixture	New extension
1.3.0	0.3.0	0.7.0	1.0.0
1.3.1	0.4.0	0.8.0	1.1.0
1.4.0	0.5.0	0.9.0	1.1.0

Each case exercised availability refresh without selection changes, explicit updates, exact update-all preview/confirmation, atomic stale-preview rejection, baseline selection rollback, separate Add, deployment of selected exact artifacts, and removal without deleting an existing deployment. Existing workload UIDs, specs, resolved image digests, and HTTP version responses were compared.

Additional executed checks include persistent selections across Core restarts, real catalog outage with retained availability, immutable SDK publication under conditional-write conflicts, rejection of payload/digest rewrites, nonadmin 403s, terminal withdrawal after omission/republication, and admin-only governance changes. The actual PostgreSQL barrier test checks selection/overlay admission ordering in an isolated schema and removes that schema afterward.

The real UI exercised Update, Choose version/baseline rollback, Add, and confirmed Update all. The read-only account sees selected/latest state without mutation controls. Release notes are omitted. Legacy/imported duplicate version labels carry distinct provenance and identity.

Existing-deployment recovery was exercised through Core's internal recovery path: a deployment originally on 0.9 was recreated while the installation selection was 0.5. It retained its database identity, frozen runtime-artifact hash, exact image and HTTP version while receiving a new Kubernetes workload UID. This is an internal recovery test; there is no public restart endpoint implied by it.

Recorded evidence checkpoints

The fresh offline installation and repeated three-version matrix passed. The matrix used the Core source commit documented above. Subsequent review fixes received targeted live checks and source-hash verification. Kamiwaza retains each checkpoint separately, together with a machine-readable qualification evidence file (`receipt.json`). Final runtime checkpoint `82a81921f8a4152c0f404ad97f51fbf274079229` verifies 44 runtime/schema files and 20 frontend files, with targeted routing, failure rollback, authorization, migration and legacy-refresh checks. All 15 proof workloads remained unchanged; all 24 deployments were ready. The full matrix is not relabeled as a later-commit run. Credentials, tokens and private database backups are excluded from those artifacts.

Known validation limits

These tests exercise purpose-built extension fixtures on an isolated instance. License enforcement was disabled in the isolated test installation. These tests do not certify third-party extension behavior, data migrations, or actual released Core builds. The publisher preserves immutable catalog payloads and uses digest-qualified artifacts; registry operators must retain those artifacts.

OmniParse Service Guide

OmniParse is the parsing and transcription extension used by Kamiwaza's context manager for media-heavy ingestion flows. When enabled, it handles file types that need richer extraction than the built-in document parsers can provide, especially audio, video, and image-oriented inputs.

What It Does

OmniParse is the public-facing parsing surface behind:

- audio and video transcription routed through the context manager
- OCR- and vision-oriented extraction for supported media types
- advanced parsing flows where the caller sets `use_omniparse=true`

For DDE-specific ingestion behavior and job flow, see Distributed Data Engine (see online documentation).

Supported Media Types

The current context-manager configuration includes these OmniParse-backed media extensions:

- audio: `.aac`, `.aif`, `.aiff`, `.flac`, `.m4a`, `.m4b`, `.mp3`, `.oga`, `.ogg`, `.opus`, `.wav`, `.wma`
- audio-from-video: `.mov`, `.mp4`, `.webm`
- images and rich documents: `.bmp`, `.docx`, `.jpeg`, `.jpg`, `.md`, `.pdf`, `.png`, `.tiff`, `.txt`

The exact list comes from the platform's current `omniparse_supported_file_types` setting.

Common Configuration

These are the main public knobs that affect OmniParse behavior in the current release:

- `use_omniparse=true` enables OmniParse-backed extraction for a request or pipeline run.
- `strict_omniparse=false` allows fallback to built-in extractors where one exists instead of failing immediately on an OmniParse error.
- `CONTEXT_MANAGER_MAX_FILE_SIZE` controls the default decoded file-size ceiling for context-manager uploads. The default is `100 MB`.
- Streaming uploads use a `200 MB` ceiling by default.
- OmniParse request timeout defaults to `300` seconds.

Operational Notes

- OmniParse is invoked by the context manager; end users typically encounter it through upload and ingestion workflows rather than by calling the extension directly.
- Successful transcription or extraction is still required to produce chunks. Empty transcription results behave like empty content for indexing.
- For extension-runtime and auth guidance, keep the developer-facing flow in Developer Guide (see online documentation) and the public workroom/session contract in Workroom Runtime Contract (see online documentation).

Kaizen User Guide

Welcome to **Kaizen**, an AI agent platform built by Kamiwaza. Kaizen lets you create AI agents that go beyond simple chat. They can execute code, search the web, connect to corporate tools like Microsoft 365, and produce polished deliverables like charts, PDFs, and presentations.

Instead of writing code to configure agents, you get a visual interface. Instead of terminal output, you get a real-time chat where you can watch your agent work and download what it produces.

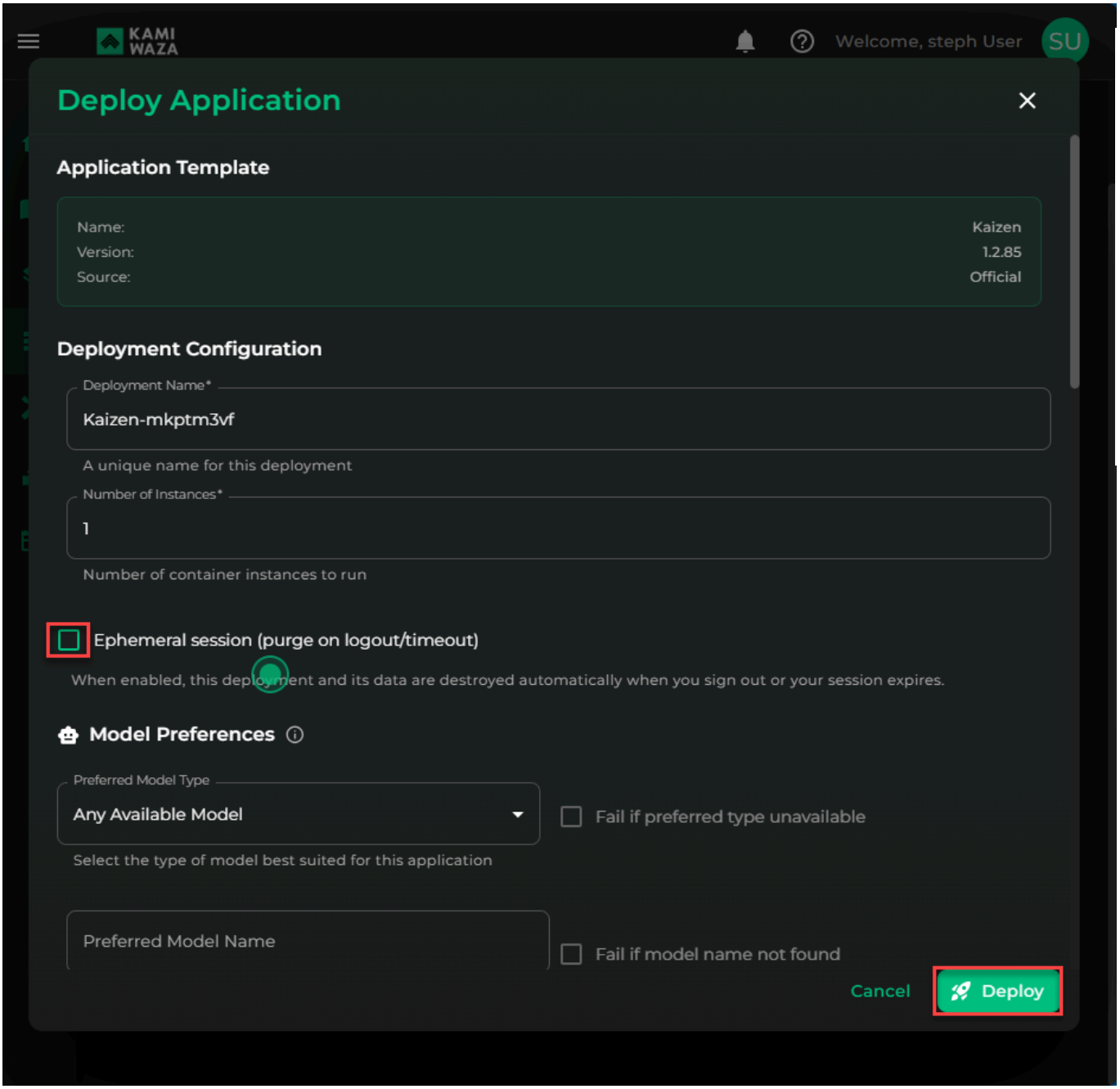
If you launch Kaizen from the **Workrooms** page (or from the deprecated Workroom Manager extension, on installations that still use it), the conversation runs inside that workroom context. That lets analysts keep their agent activity aligned to a specific mission or case while still using the same Kaizen workflows described here.

In a shared non-global workroom, Kaizen agents, conversations, uploads, and generated outputs are visible to authorized workroom members according to role. Owners and Contributors can continue shared conversations, while Viewers can open them read-only. The **Global Workroom** keeps legacy personal visibility behavior and is not treated as a shared Kaizen workspace.

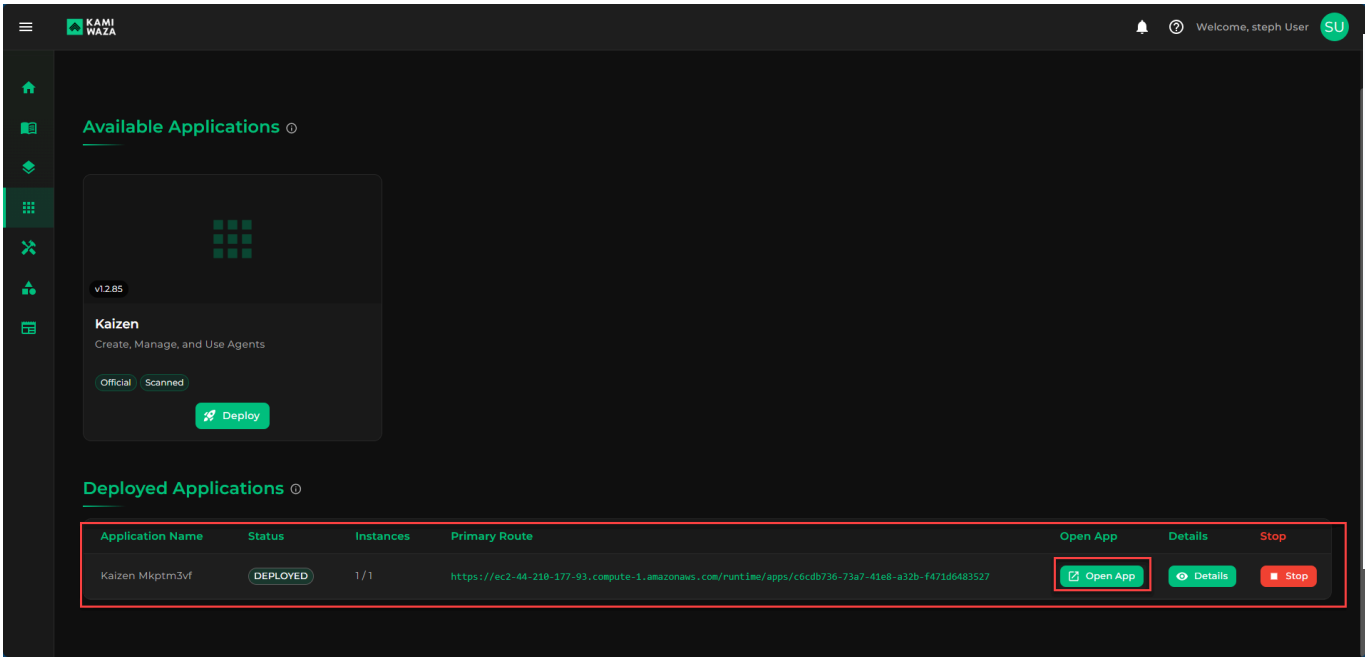
Launch Kaizen

Kaizen is available as an extension in **App Garden**.

1. Open **App Garden** in the Kamiwaza UI.
2. Locate the **Kaizen** app and click **Deploy**.
3. Choose whether you want an **Ephemeral Session**. Leave the default settings unless you have a specific reason to change them.

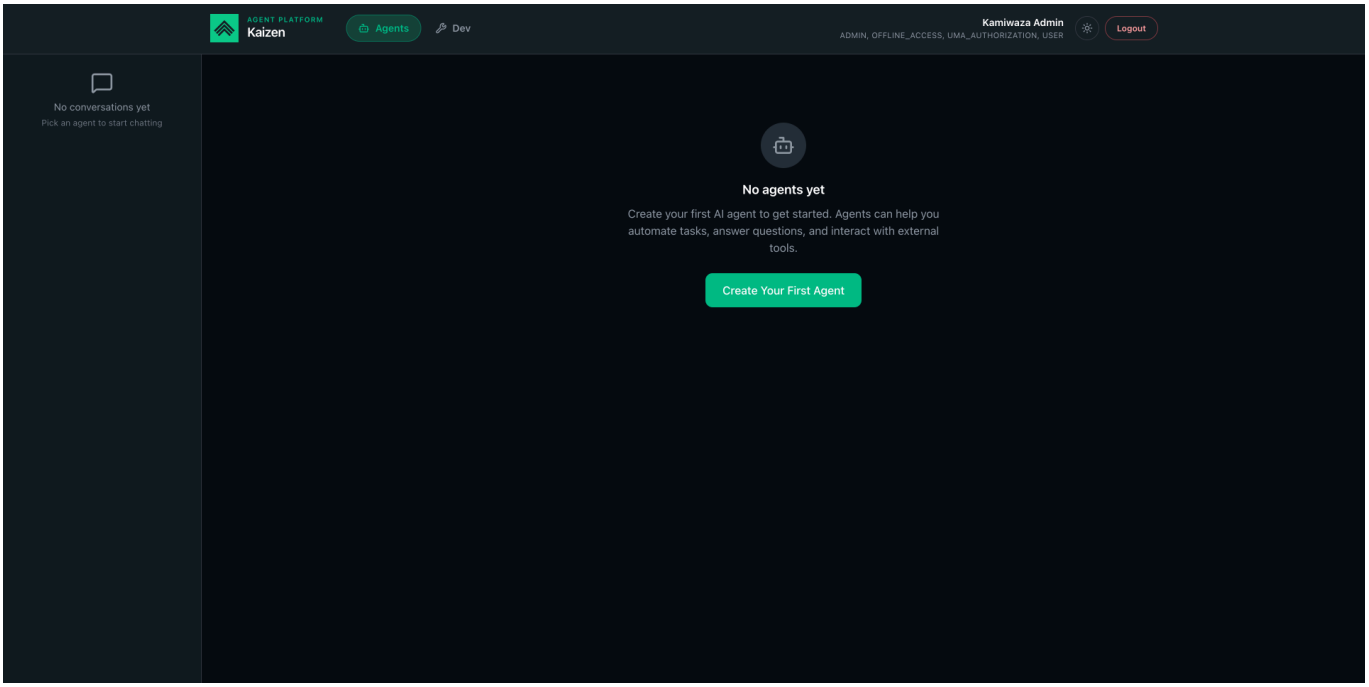


4. Click **Deploy**.
5. Once the deployment is ready, click **Open App**.



Getting Started

When you first open Kaizen, you'll see an empty Agent Library. This is your home base, where all your agents live and where you launch conversations.



Each agent you create is a persistent configuration. Choose its AI model, attach skills and tools, and set a security policy. Then start as many conversations as you need.

Click **Create Your First Agent** to get started.

The top navigation bar provides access to all sections of Kaizen:

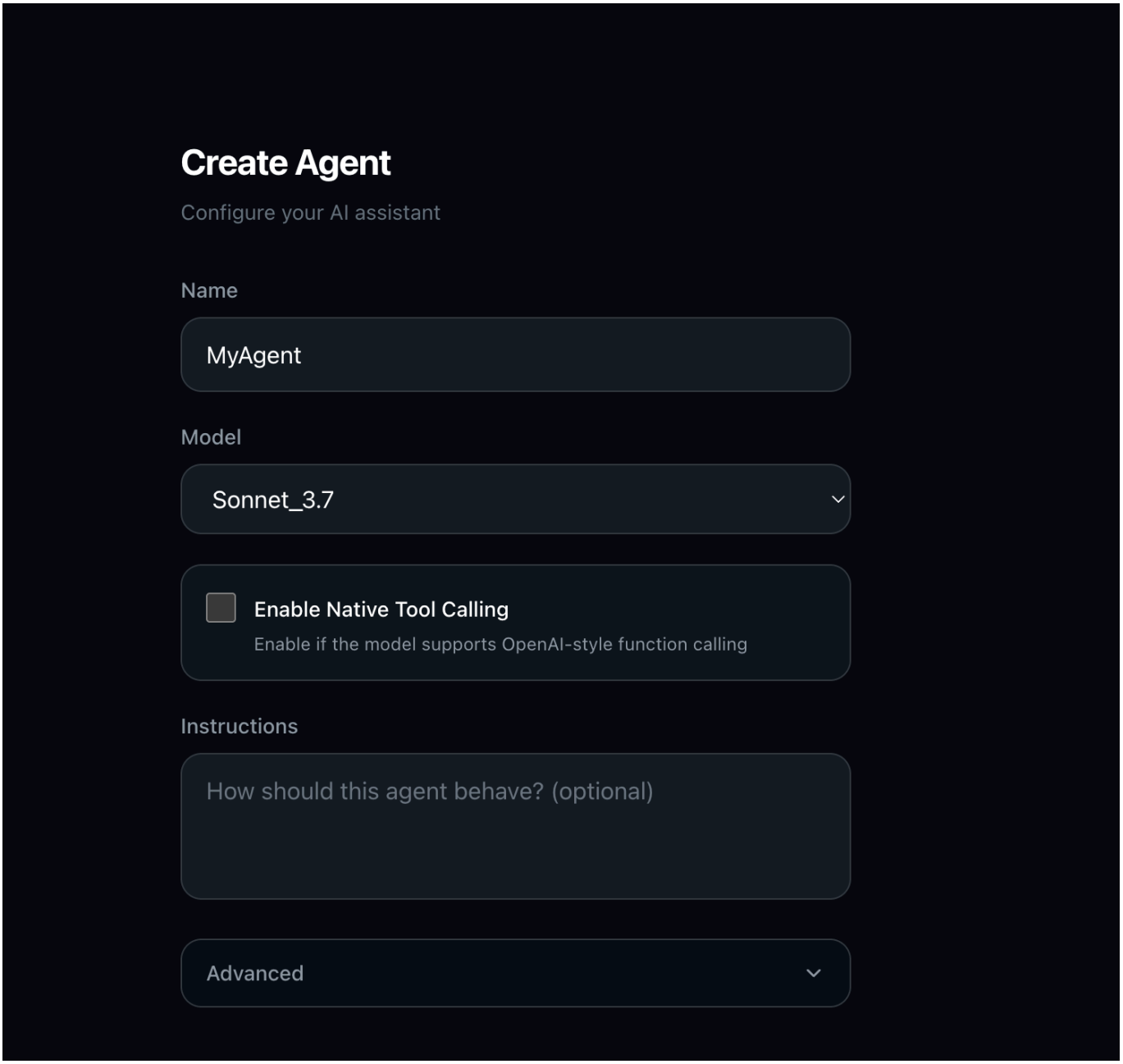
- **Agents** — create and manage your AI agents (home view)
- **Conversations** — browse and filter past conversations across all agents
- **Search** — search across conversation messages, tool calls, and results
- **Metrics** — model health and inference metrics for your Kaizen environment

---

Creating Your First Agent

The agent builder walks you through configuration in six steps.

Step 1: Basic Configuration



- **Name**: Give your agent a descriptive name for its purpose, such as "Data Analyst" or "Report Writer".

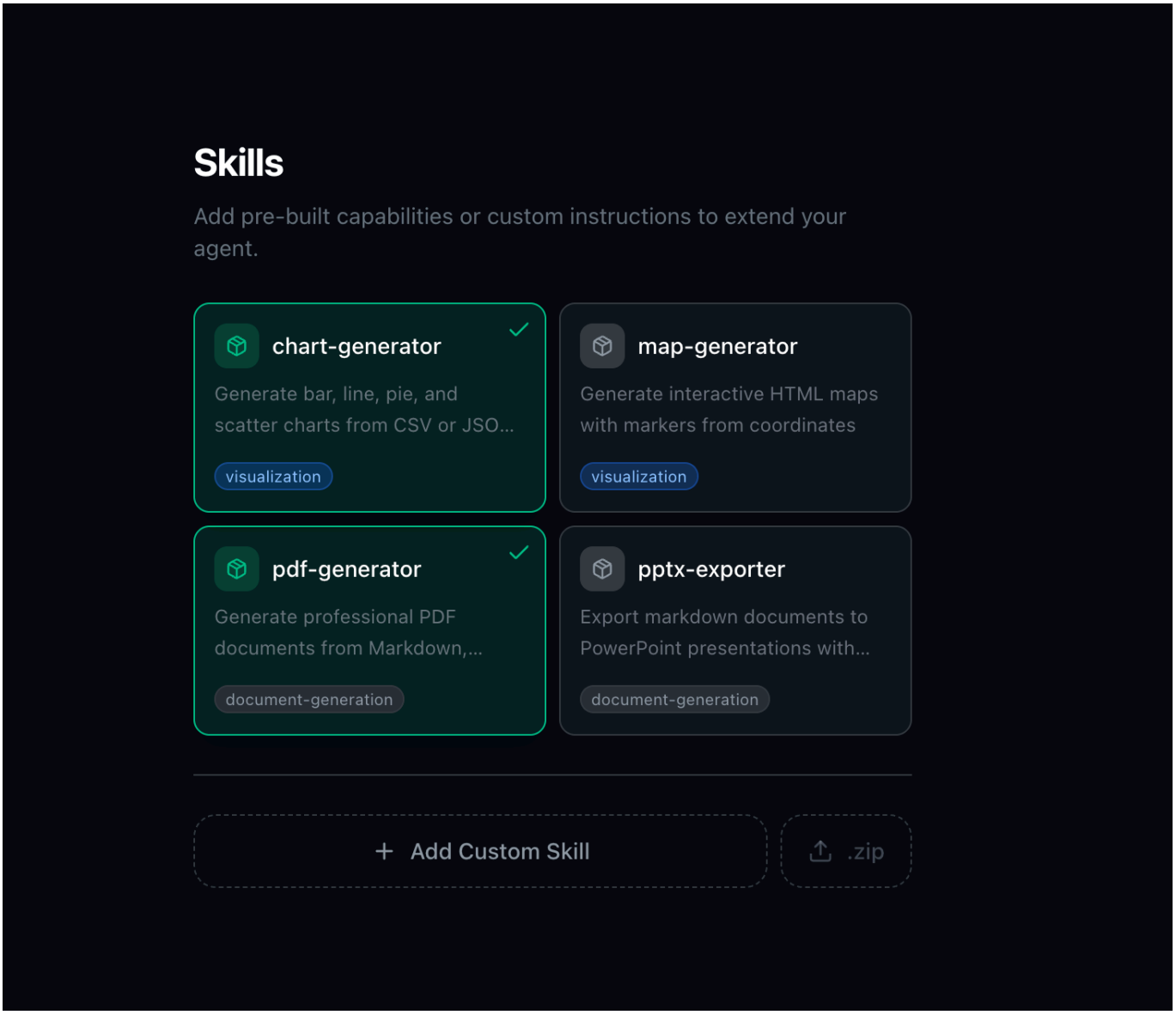
- **Model**: Select an AI model deployed on your Kamiwaza instance. Models are managed by your admin through the Kamiwaza Model Library.
- **Native Tool Calling**: Enable this if your model supports OpenAI-style function calling.
- **Instructions**: Add optional instructions to shape how the agent behaves, such as domain expertise, tone, or policies.

> **Tip:** Use Instructions to give your agent a specific role. Example: *"You are a financial analyst. Always format currency with two decimal places and cite data sources."*

Step 2: Adding Skills

Kaizen uses the AgentSkills model. In practice, you can give an agent skills in two ways:

- **From Skills Library**: Attach published, reusable skills from your shared library.
- **Custom Skills**: Create agent-specific skills directly in Kaizen.



Library skills are best for reusable capabilities your team wants to standardize across many agents. Custom skills are best for local instructions, business context, and lightweight workflows specific to one agent.

Custom skill trigger options:

- **Always Active**: The skill is always available to the agent.
- **Keyword Triggered**: The skill activates when matching keywords appear in the conversation.
- **Task Triggered**: The skill activates from explicit commands or workflow phrases and can include structured inputs.

> **Tip:** Start with a few small custom skills for instructions and domain context, then attach shared library skills for heavier packaged capabilities your team wants to reuse.

Step 3: Attaching Packaged Skills

Many advanced capabilities are delivered as packaged skills from the Skills Library rather than a separate "bundle" system. You can browse the library and attach the skills your agent needs.

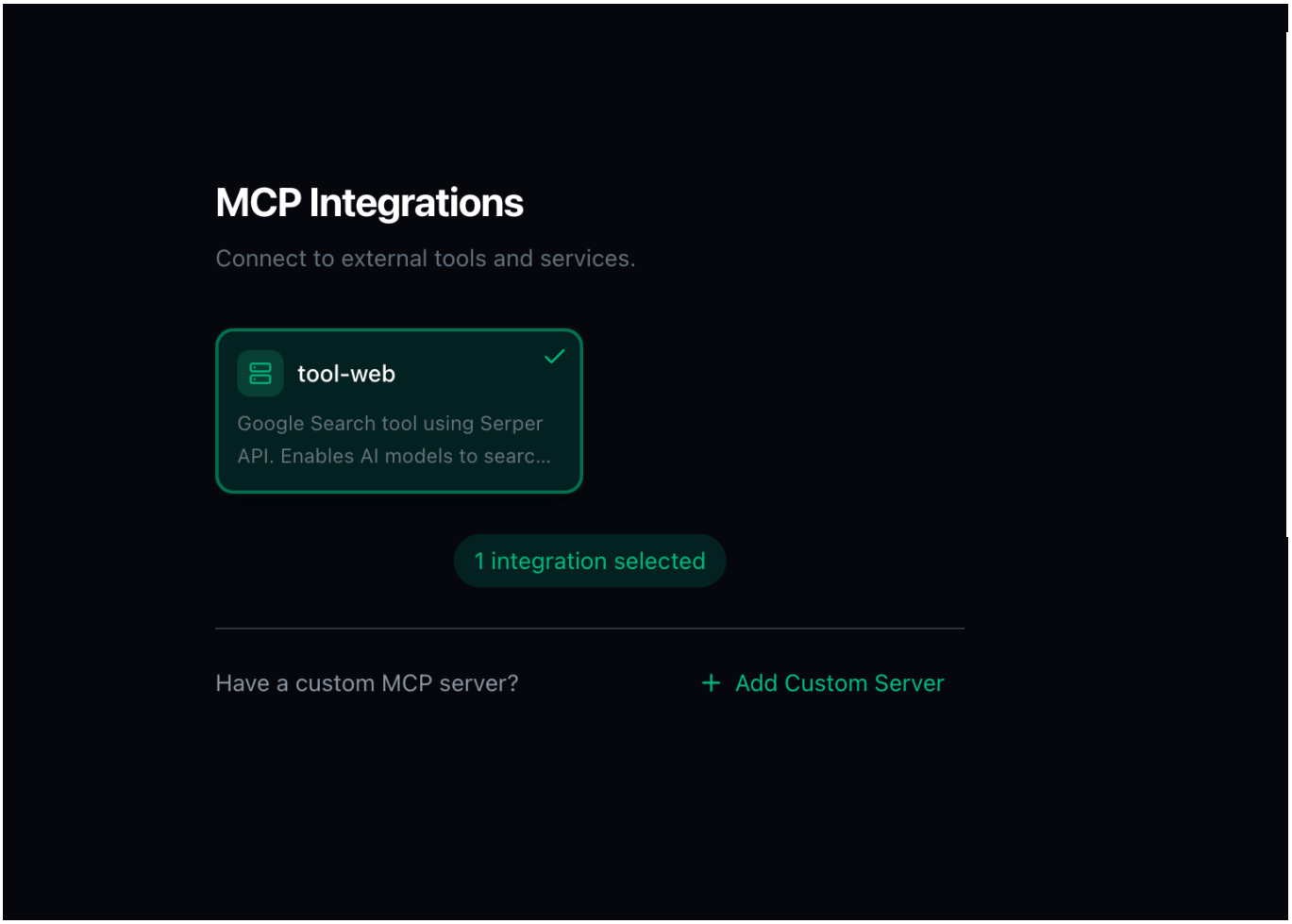
Common examples include skills for:

- **Charts and visualizations**
- **Maps and geospatial outputs**
- **PDF generation**
- **PowerPoint export**
- **Document ingestion and context processing**

Some Kaizen environments also include built-in default skills automatically, so your agent may already have baseline capabilities without any extra setup.

Step 4: Connecting Tools (MCP Integrations)

MCP (Model Context Protocol) integrations give your agent access to external tools and services such as web search, APIs, databases, and more.



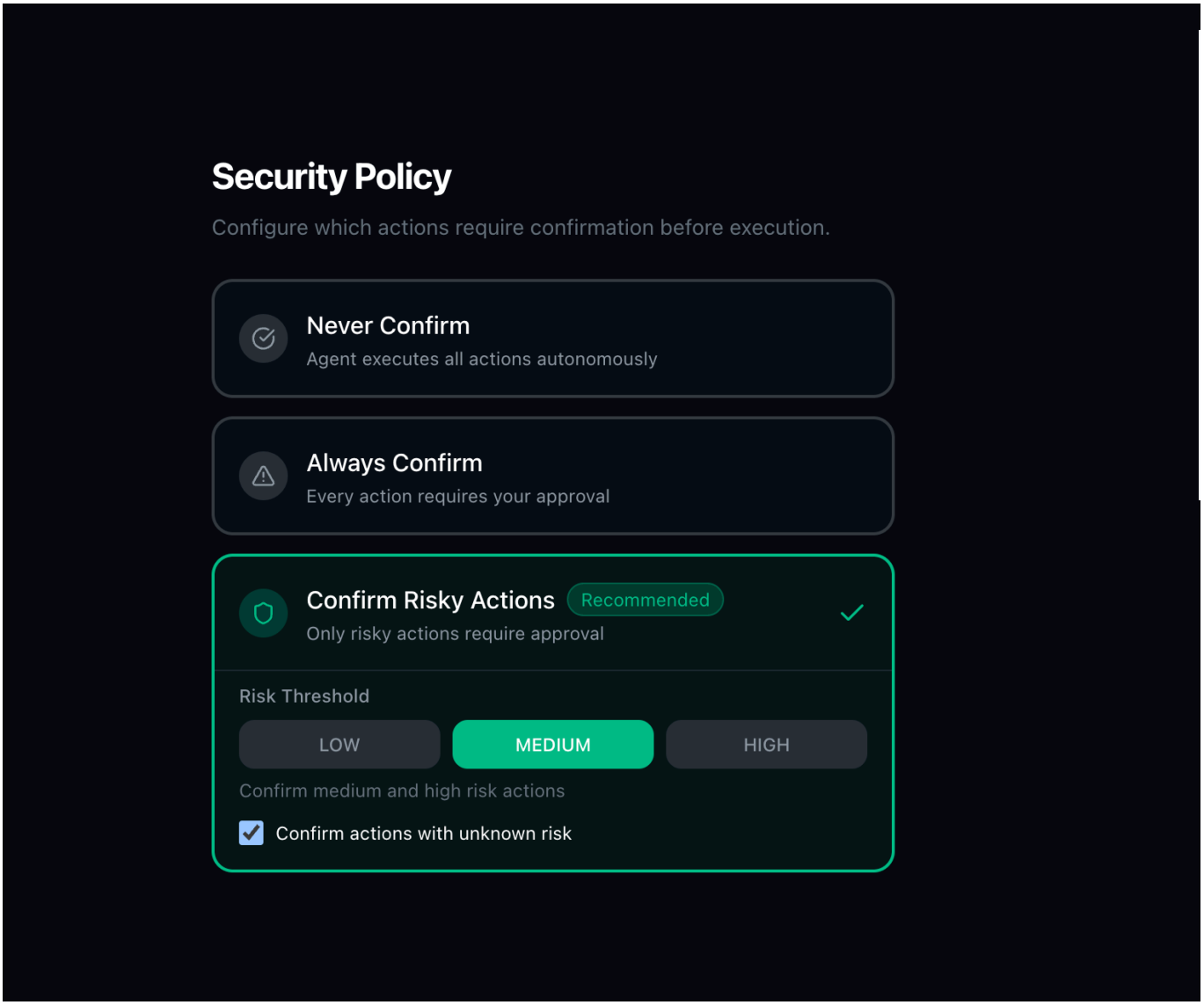
Tools deployed through the Kamiwaza Tool Garden are automatically available for selection here. Your admin deploys tools to the platform, and they appear ready to attach to your agents.

You can also connect custom MCP servers for proprietary or third-party APIs using **+ Add Custom Server**.

> Without tools, your agent can only reason and write code. With tools, it can search Google, query databases, call REST APIs, interact with SaaS platforms, and much more within a single conversation.

Step 5: Security Policy

Control how much autonomy your agent has. The security policy determines which actions require your approval before the agent executes them.



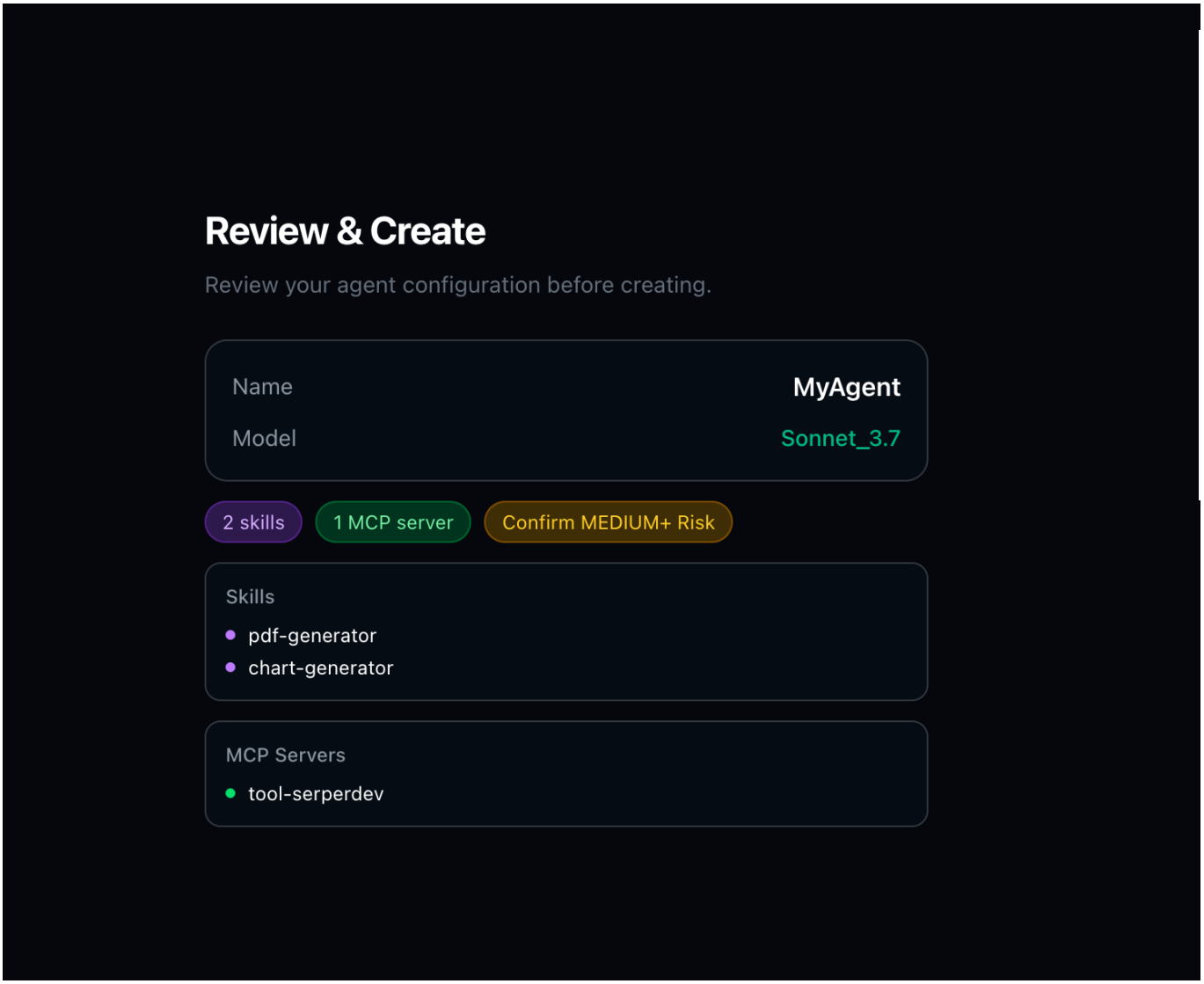
Three modes:

- **Never Confirm**: The agent executes all actions autonomously. Best for trusted, low-risk tasks.
- **Always Confirm**: Every action requires your approval.
- **Confirm Risky Actions** (Recommended): Only actions above your chosen risk threshold need approval. The agent handles routine tasks on its own.

The **Risk Threshold** (`Low`, `Medium`, or `High`) determines the cutoff. With `Medium` selected, the agent asks for confirmation on medium- and high-risk actions but proceeds autonomously on low-risk ones. You can also choose whether actions with unknown risk levels require confirmation.

Step 6: Review & Create

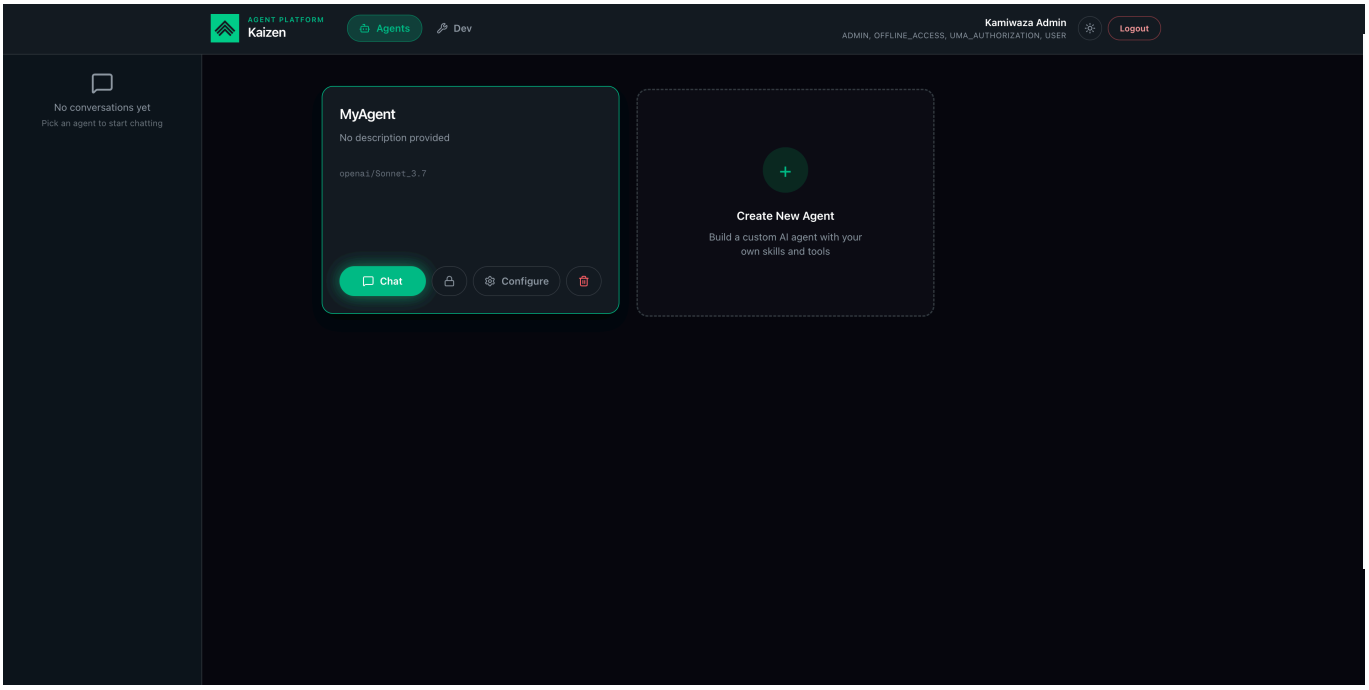
Before creating your agent, review the full configuration summary. Verify your model, custom skills, attached library skills, connected tools, and security policy.



The summary badges give you an at-a-glance view of custom skills, library skills, MCP servers, and security policy. Once satisfied, click **Create Agent**.

Your Agent Library

After creation, your agent appears in the library. Each agent card shows its name, model, and quick actions.



- **Chat**: Start a new conversation with this agent.
- **Private Session**: Start an ephemeral conversation that is automatically deleted when you leave. This is available in personal/global scope and is intended for sensitive or one-off tasks.
- **Configure**: Edit the agent's model, skills, bundles, tools, or security policy.
- **Delete**: Remove the agent and its conversations.

Create as many agents as you need, such as a research agent, a data analyst, or a report writer. Each agent keeps its own configuration. In a shared non-global workroom, that agent and its conversations are visible to other authorized workroom members, while the creator remains recorded for attribution and auditing. Shared visibility does not automatically grant reuse of the creator's private credentials; collaborator runs continue to use the acting user's own credentials or an explicitly shared workroom credential path.

The left sidebar shows recent conversations across all visible agents. In a shared non-global workroom, that means shared workroom history by default, with a list-level filter for items created by you. Click any conversation to resume where work left off.

Using Your Agent

Sending Messages

Click **Chat** on any agent to start a conversation. Type your request in the message input and the agent will get to work. You can watch its progress in real time as it executes code, calls tools, and produces results.

Uploading Files

Click the **+** button in the message input to attach files. You can upload from your local device or pull files directly from **Microsoft 365** (OneDrive, SharePoint).

> **Note:** The Microsoft 365 connector must be configured by an administrator in the Kamiwaza UI before it is available to users. Once configured, you can browse and attach OneDrive and SharePoint files directly in your conversations.

When Kaizen is opened from a workroom, use these uploads and connected sources to support that workroom's analysis workflow.

Grounded Answers and Citations

Depending on your deployment and the tools attached to your agent, Kaizen can produce grounded answers that include citations back to source material such as uploaded files, connected document stores, or web results.

To get the best results:

- attach the files or tools your agent should use before asking the question
- ask explicitly for citations when you need traceable answers
- verify that the cited sources actually support the claim before sharing the output

If a response should be grounded but comes back without citations, check whether the relevant data source, skill, or retrieval path is configured in your environment.

Related Data Suggestions

Current Kaizen builds can show a **Related Data Suggestions** card around an active conversation. This panel aggregates likely useful:

- workspace files
- published or attached skills
- Microsoft 365 results

Each suggestion includes a score and a short reason so you can decide whether to pull that source into the conversation. Kaizen keeps partial results visible even if one source is degraded. For example, if an M365 connector is unavailable or a search times out, the panel can still show file or skill suggestions together with a diagnostic note instead of failing the entire suggestion pass.

Use this panel when you want to ground the next turn, find the right file faster, or understand why Kaizen thinks a specific skill or document may help with the current task.

Downloading Results

Any files created by the agent, including reports, charts, code, and presentations, appear in the workspace panel on the right side of the chat. You can preview and download them directly.

Preview behavior depends on file type. In current releases, Kaizen can render:

- presentations as slide previews
- DOCX files as extracted inline text
- images such as SVG and WEBP inline in the browser
- CSV and Excel files in a table view

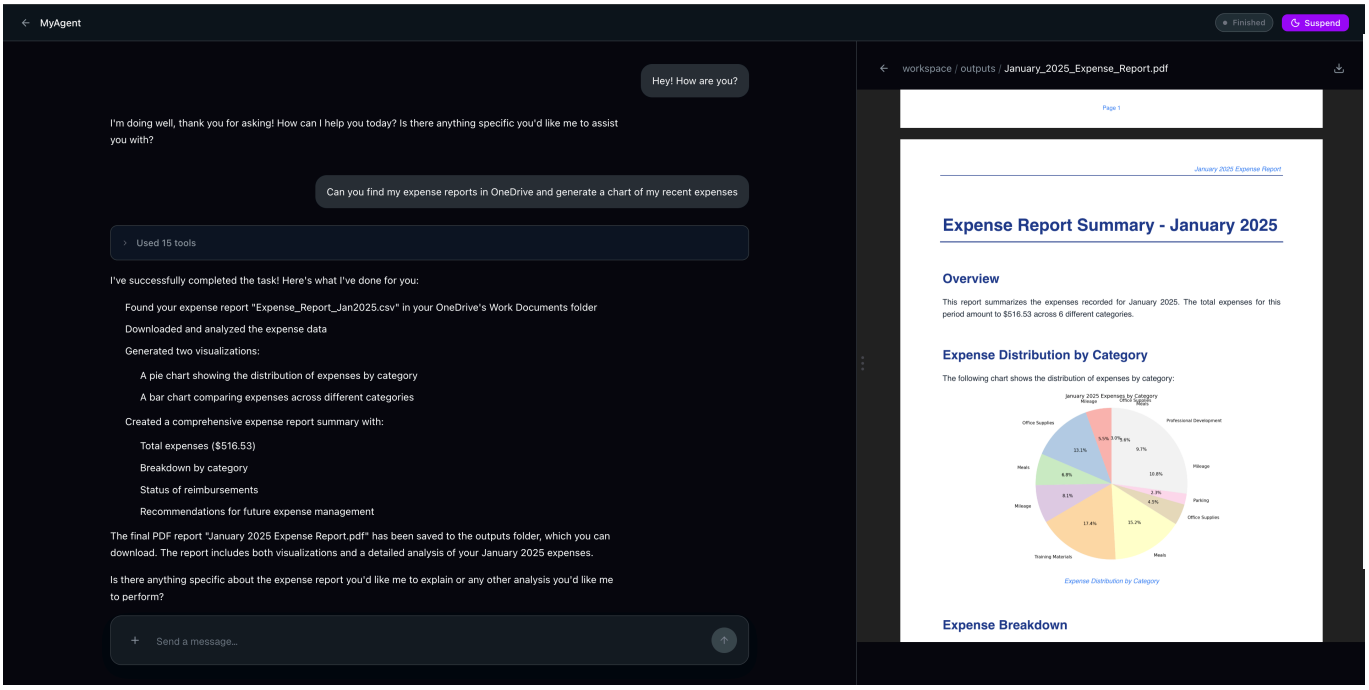
If a file type is not previewable in place, use the same panel to download it.

Your Agent in Action

Here is a real example: a user asked their agent to find expense reports in OneDrive and generate a chart and summary PDF. The agent:

- found the expense report CSV in OneDrive
- analyzed the expense data
- generated a pie chart and bar chart
- created a professional PDF report with the analysis

The generated PDF is viewable and downloadable directly in the right panel.



Conversations

The **Conversations** tab gives you a dedicated view of all visible conversations across every agent.

Use the filters at the top of the page to narrow down what you're looking for:

- **Filter by Agent** — select a specific agent to see only its conversations, or choose All Agents.
- **Visibility** — switch between all workroom-visible conversations and only the ones you created.
- **Sort By** — order conversations by most recent or other criteria.

In a shared non-global workroom, this view is designed for handoff as well as recall. Click any conversation to resume where work left off, even if another Owner or Contributor created the earlier turns.

Search

The **Search** tab lets you search across all your past conversations to find specific messages, tool calls, and results.

Searching Conversations

Enter a search term to find matching content across all conversations. You can refine results with filters:

- **Tool Name** — filter by a specific tool (e.g., web search, code execution, file access).
- **Event Type** — filter by the type of event (messages, tool calls, results).
- **Date Range** — narrow results to a specific time window using the From and To date fields.

Click **Search** to run your query. Results show matching content with context so you can quickly locate what you need.

Recipes

The **Recipes** tab (next to Search) lets you save useful tool call patterns that you discover while searching. Recipes capture a reusable pattern so you can reference it later without having to search for it again.

To create a recipe, find a tool call pattern in your search results and save it. Your saved recipes appear in the left sidebar for quick access.

Recipes are useful for:

- Capturing effective tool call sequences that produced good results
- Building a personal library of proven patterns for common tasks
- Sharing approaches across sessions without recreating them from scratch

---

Metrics

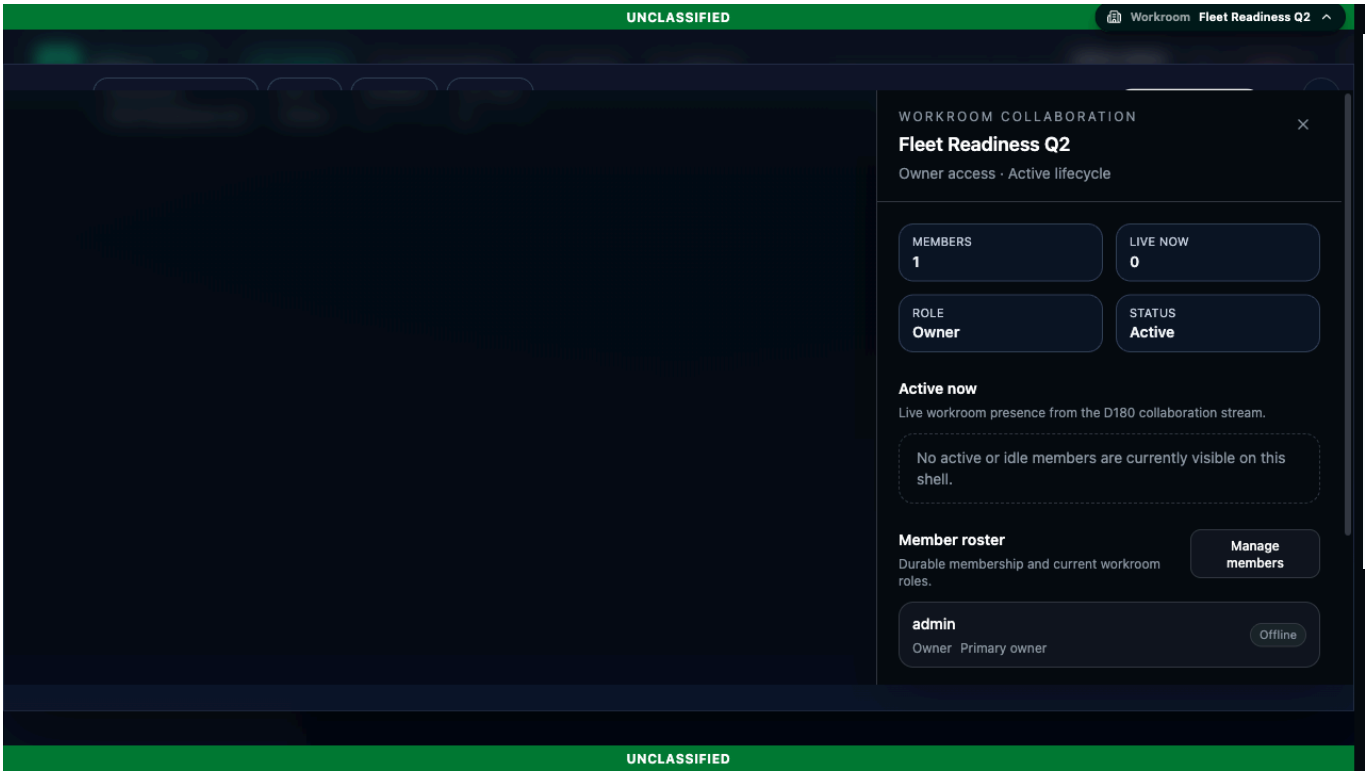
The **Metrics** tab shows model health information for your Kaizen environment.

The view displays inference performance and model availability metrics. Use the time range selector (**24h**, **7d**, **30d**) to adjust the reporting window.

The Metrics view is most useful once you have active agents running conversations and processing data.

Workroom Collaboration

When Kaizen is launched from a workroom, a **workroom info bar** appears at the top of the page showing the workroom name and classification banner. Click it to expand the workroom status summary.



The expanded bar shows:

Field	Description
Workroom	The name of the workroom this Kaizen session belongs to.
Role	Your role in the workroom (Owner, Contributor, or Viewer).
Members	Total number of members with access to this workroom.
Live Now	How many members are currently active in the workroom.

Click **Collaboration** to open the full collaboration panel on the right side. The panel includes:

Active Now

Shows which members are currently active or idle in the workroom. This is a live view from the collaboration stream — use it to see who else is working in the same workroom at the same time.

Shared History

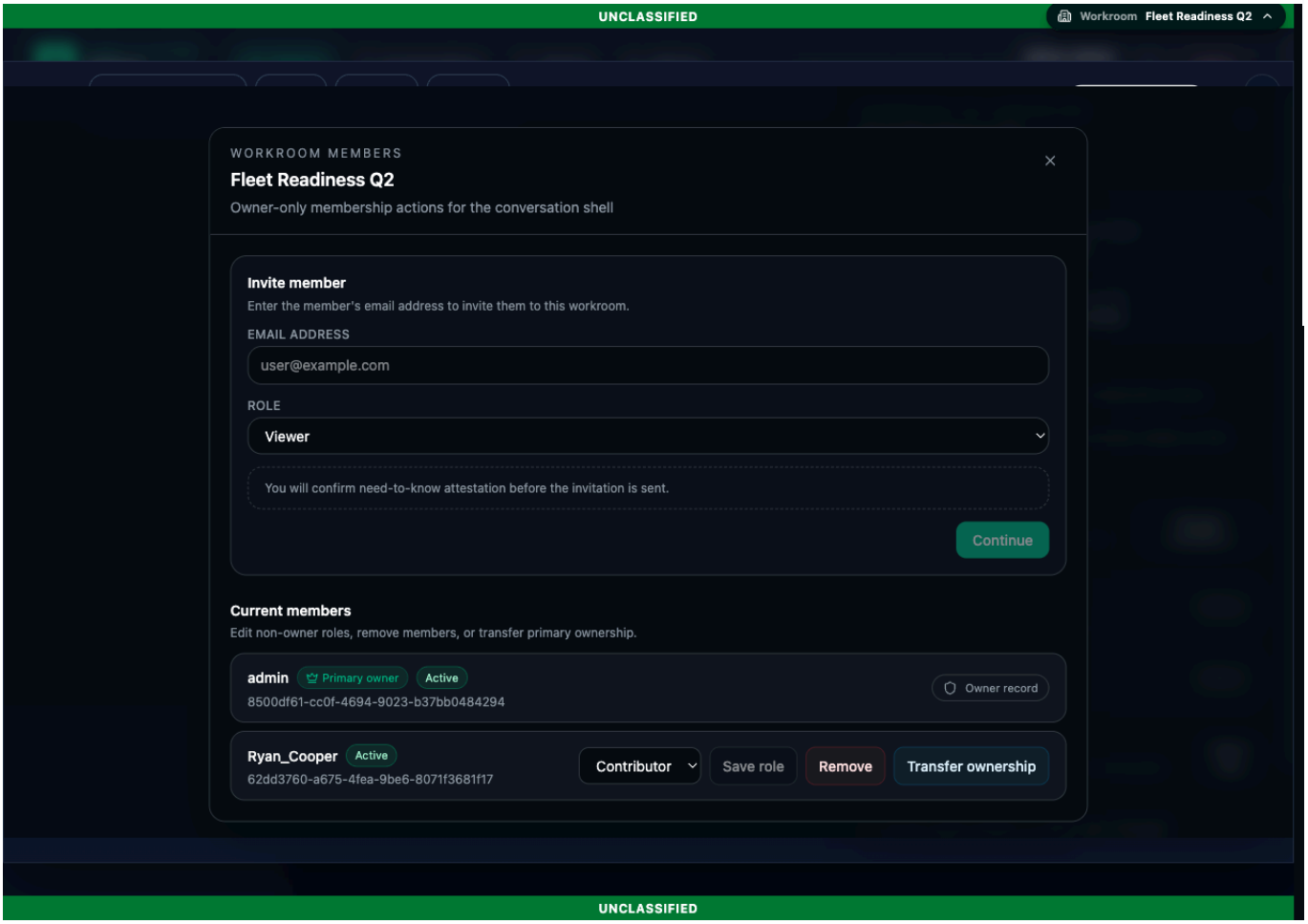
In a shared non-global workroom, Kaizen keeps conversation history as shared workroom history. Later Owners and Contributors can open the same thread and continue it without losing prior context. Viewers can read that history but cannot add turns, upload files, or mutate agent settings.

Turn Attribution

Kaizen attributes each human or agent-triggering turn to the acting user who initiated it. That attribution stays with the turn for later handoff and review; it is not inferred from the original conversation creator.

Member Roster

Lists all workroom members with their roles and online status. Owners can click **Manage members** to open the membership management panel from within Kaizen. The **Workrooms** page (or the deprecated Workroom Manager extension, on installations that still use it) is the standard place for workroom administration, including sharing, role changes, member removal, and ownership transfer.



The **Workroom Members** panel lets owners:

- **Invite a member** — enter an email address, select a role (Viewer or Contributor), and confirm the attestation: *"I confirm the invited user has need-to-know for this workroom's content."*
- **Change a member's role** — use the role dropdown on any non-owner member and click **Save role**.
- **Remove a member** — click **Remove** to revoke access. If the member has an active session, you may be asked to confirm.
- **Transfer ownership** — click **Transfer ownership** to make another member the primary owner of the workroom.

Audit Log

Tracks recent lifecycle, membership, credential, and access activity for the workroom. Click **Open log** to view the audit timeline. Use the audit log to verify who changed membership, delegated credentials, or modified workroom lifecycle state during the current collaboration window.

Shared Credentials

Shared visibility does not automatically grant access to another member's private credentials. If a shared agent depends on a connector, API key, or OAuth session, the acting user must use their own credentials or an explicitly workroom-shared credential binding when that capability is enabled in your environment.

Agent Capabilities

Capability	Description
Generate Documents	Create PDFs, PowerPoint presentations, and structured reports from your data or instructions.
Create Visualizations	Bar charts, line charts, pie charts, scatter plots, and interactive maps from your data.
Search the Web	Find current information, research topics, and gather data using connected search tools.
Access Corporate Files	Browse and analyze files from OneDrive and SharePoint through the Microsoft 365 integration.
Analyze Data	Process CSV files, run calculations, identify trends, and produce statistical summaries.
Connect to APIs	Interact with external services through MCP tool integrations, including CRMs, databases, and SaaS platforms.
Code Execution	Run commands and scripts in an isolated container environment.
Real-time Streaming	Watch the agent's thought process and terminal output live.
Security Policies	Configure sensitive-action controls with risk-based confirmation.

Tips for Success

- **Start Simple**: Create your first agent without complex skills or tools, then add them as needed.
- **Be Specific**: Clear, detailed prompts yield better results.
- **Use the Skills Library**: Attach published skills for common outputs like charts, PDFs, presentations, and document processing.
- **Check Connections**: If you use MCP tools, verify connections before saving.
- **Monitor Status**: Watch for "Waiting for Confirmation" states if you have strict security policies.

Example Prompts

- *"Create a Python script that analyzes this CSV data and generates a summary report."*
- *"Find my expense reports in OneDrive and generate a chart of spending by category."*
- *"Research the latest trends in renewable energy and create a PowerPoint presentation."*
- *"Analyze this dataset and create a PDF report with visualizations."*

Troubleshooting Kaizen

The agent is waiting for confirmation

Your agent's security policy may require approval before it can run tools or commands. Approve the pending step, or update the policy in the agent configuration if the current threshold is too strict for the workflow.

I cannot access Microsoft 365 files

The Microsoft 365 connector must be configured by an administrator before it appears to end users. If the option is missing or authentication fails, confirm the platform configuration with your administrator.

I expected citations, but none appeared

Make sure the agent has access to the documents or tools it needs, and ask for citations explicitly when traceability matters. If the issue persists, verify the data source or retrieval configuration in your deployment.

I need logs for a failed deployment or broken conversation

Start with the Kaizen deployment logs in the Kamiwaza log viewer, then review the platform-level guidance in Observability (see online documentation). That is the fastest path for diagnosing extension startup failures, connector issues, or model-side errors.

Learn More

- Models Overview (see online documentation)
- Tool Garden (see online documentation)
- Observability (see online documentation)

Kaizen Changelog

This document tracks major changes, feature additions, and improvements in the Kaizen agent platform.

Version 1.4.0

Major Features

1. Skills-First Agent Workflows

- **Reusable skills**: Expanded Kaizen's built-in skill system with richer prebuilt capabilities, including chart, map, PDF, PPTX, and skill-builder workflows.
- **Skills over bundles**: Continued the shift away from older bundle-centric behavior toward reusable, agent-native skills and defaults.
- **Skill visibility**: Improved agent creation and conversation UX so active skills and skill configuration are easier to inspect and manage.

2. Context-Aware Document Processing

- **Context Service integration**: Added first-class support for collections, pipeline jobs, retrieval, and semantic search through Kamiwaza Context Service.
- **Ingestion workflows**: Agents can submit files for processing, monitor progress, and use retrieved context in downstream work.
- **Operational guardrails**: Automatic context-search tool injection was temporarily removed while internal authentication issues are stabilized.

3. Better Model, Tool, and Enterprise Integrations

- **Kamiwaza model access**: Kaizen now centers on Kamiwaza-managed model deployments and custom endpoints instead of direct Anthropic or Claude-specific setup.
- **Tool Garden and MCP improvements**: Improved Tool Garden MCP discovery, authentication, PAT fallback, host resolution, and error handling.
- **Microsoft 365 and SharePoint**: Expanded M365 and SharePoint browsing, download handling, schemas, and timeout behavior for more reliable enterprise file access.

4. Workroom-Aware and App Garden-Ready Deployment

- **Workroom preservation**: Kaizen now preserves workroom context across deployments and related API calls.
- **App Garden compatibility**: Improved App Garden deployment compatibility, runtime defaults, and route/base-path behavior in packaged deployments.
- **Release metadata**: Updated extension metadata, preview image handling, and compatibility to target Kamiwaza `>=0.10.0`.

5. Sandbox and Runtime Controls

- **LLM tuning controls**: Added configurable timeout, retry, and consumer-hardware defaults for agent model calls.
- **Sandbox storage options**: Added configurable sandbox storage sizing and persistence controls for workspace durability.
- **Runtime hardening**: Improved Kubernetes sandbox/runtime support, health checks, and container security context handling.

6. UX and Reliability Improvements

- **Conversation experience**: Improved approval cards, error toasts, file previews, citations, and conversation reliability under failure conditions.
- **File handling**: Added support for image attachments and larger uploads, with frontend middleware updates for bigger payloads.
- **Workspace polish**: Improved agent creation wizard layout, workspace panels, and context-processing progress surfaces.

Technical Improvements

- **Security**: Hardened error sanitization, upstream error handling, TLS/SSL behavior, and container/runtime defaults.
- **Platform resilience**: Added platform event logging, HTTP pool improvements, better health endpoints, and more robust SSE/conversation handling.
- **Dependency maintenance**: Upgraded vulnerable dependencies, refreshed SDK patches, and improved test coverage across context, MCP, runtime-default, and sandbox flows.

Version 1.3.2 vs 0.9.0

🚀 Major Features

1. App Garden Integration

- **Deployment**: Now fully deployable as a Kamiwaza App Garden extension.
- **Configuration**: Auto-generated `docker-compose` and metadata for seamless platform integration.
- **Streaming**: Migrated from WebSockets to **Server-Sent Events (SSE)** for better compatibility with corporate proxies and HTTP-only environments.

2. Advanced Skills System

- **Custom Knowledge**: Inject domain expertise into agents.
- **Skill Types**:
  - **Repository Skills**: Always-active context (policies, business rules).
  - **Knowledge Skills**: Triggered by specific keywords.
  - **Task Skills**: Pre-defined workflows with inputs.

3. MCP (Model Context Protocol) Integration

- **External Tools**: Connect agents to external APIs and services (Zapier, Google Workspace, Slack).
- **Security**: Encrypted storage for authentication headers and keys.
- **Management**: UI for configuring and testing MCP server connections.

4. Enterprise Security & Governance

- **Action Confirmation**: Configurable policies for tool execution:
  - *Never Confirm*: Autonomous execution.
  - *Confirm Risky*: Pauses for high-risk actions (file deletion, network calls).
  - *Always Confirm*: Manual approval for every step.
- **Sandbox Controller**: Optional secure mode for strict Docker isolation (Phase 8.5).

5. Enhanced Workspace & File Management

- **File Uploads**: "Paperclip" button to inject files directly into the agent's workspace.
- **File Downloads**: Browse and download generated files (reports, code) from the side panel.
- **Activity Stream**: Real-time "Thinking", "Tool Usage", and "Result" blocks for full visibility into agent logic.

6. UX Improvements

- **ChatGPT-Style Sidebar**: Time-grouped conversation history (Today, Yesterday, Last 7 Days).
- **Inline Confirmation**: Approve or reject agent actions directly within the chat stream.
- **Stuck Detection**: If an agent gets stuck in a loop or repetitive behavior, Kaizen automatically detects this and can help recover, ensuring your agent continues making progress on your tasks.

7. Microsoft 365 Integration

- **Cloud Connectors**: Connect directly to SharePoint and OneDrive to import documents into the workspace.
- **User-Delegated Auth**: Secure OAuth flow ensures users only access files they have permission to see.

🛠️ Technical Improvements

- **Backend**: Migrated to FastAPI + Python 3.12.
- **Frontend**: Upgraded to Next.js 15.
- **Database**: Robust PostgreSQL storage with Alembic migrations for agents, secrets, and conversation history.
- **SDK**: Updated to OpenHands SDK 1.8.2 with custom SSL patching for internal service communication.

Historical Version Summary

Version	Status	Key Focus
1.4.0	Production	Skills-first workflows, context pipelines, runtime hardening.
1.3.2	Production	App Garden deployment, SSE streaming, Polish.
1.2.0	Feature Complete	Sandbox Controller, File Uploads.
1.1.0	Security	Security Policies, MCP Integration.
1.0.0	Core Features	Skills System, Workspace Visibility.
0.9.0	MVP	Basic Chat, Docker Execution, Simple CRUD.

Skills Library User Guide

The Skills Library lets Kamiwaza administrators manage a shared catalog of reusable agent skills. From this app you can import skill packages, curate their metadata, control their lifecycle, and export bundles for sharing.

Layout

The interface has three panes:

- **Sidebar** (left) — Filter by status, category, tag, or classification.
- **Skill list** (center) — Search and sort skills. Sort options: Recently Updated, Alphabetical, Category.
- **Detail panel** (right) — View and edit the selected skill.

Importing Skills

1. Click **Add Shared Skill(s)** in the header.
2. Drag and drop a ZIP file or click to browse.
3. The app accepts both individual Agent Skills packages and Skills Library export bundles.
4. Imported skills appear as **Draft** entries.

A valid skill package contains a `SKILL.md` file with YAML frontmatter and may include `/scripts/`, `/assets/`, and `/references/` directories.

Editing Metadata

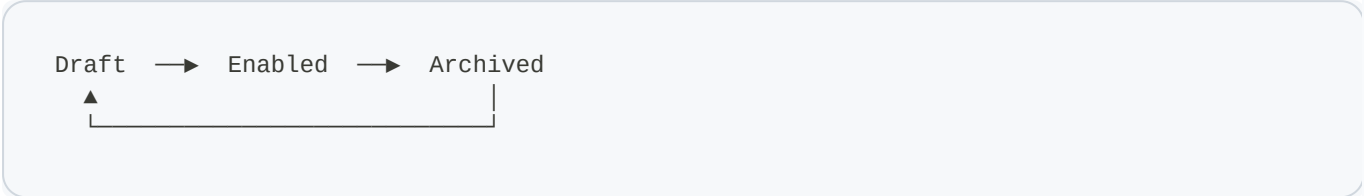
Select a skill, then click the pencil icon to enter edit mode. Editable fields:

Field	Description
Display Name	How the skill appears in the catalog
Category	Functional grouping
Classification	Security or compliance level
Catalog Tags	Comma-separated searchable keywords
Trigger	JSON defining keyword or task-based activation
Inputs	JSON defining parameters the skill accepts
Custom Metadata	Free-form JSON for extra data

Package contents (the imported ZIP) remain read-only. Click **Save** to persist changes.

Lifecycle Management

Skills move through three states:



- **Draft** — Imported but not visible to analysts.
- **Enabled** — Published and available in the Kaizen skill catalog.
- **Archived** — Hidden from the catalog but retained for history.

Transition a skill by clicking the status action button in the detail panel:

- Draft → **Enable**
- Enabled → **Archive**
- Archived → **Restore to Draft**

Searching and Filtering

- The **search box** matches against name, description, category, and tags.
- The **sidebar filters** show counts for each facet and update as you refine your selection.
- Combine search text with sidebar filters to narrow results.

Exporting Skills

1. Apply any search or filters to scope the export.
2. Click **Export** in the header.
3. A ZIP bundle downloads containing all currently visible skills.

The filename follows the pattern `skills-library-export-YYYY-MM-DD.zip`.

Deleting Skills

Open a skill's detail panel, enter edit mode, and click **Delete**. This removes the skill from the catalog. Deletion is a soft delete — the skill is hidden, not permanently destroyed.

Tips

- Use tags liberally to make skills easy to discover.
- Review imported trigger and input JSON before enabling a skill — these control how Kaizen activates it.
- Export regularly as a backup before making bulk changes.
- Toggle between light and dark themes using the header control.

Graphiti Service Guide

Graphiti is Kamiwaza's temporal knowledge graph service for agent memory, contextual retrieval, and fact-oriented search. It accepts messages, extracts entities and relationships, and returns graph context for downstream agent or RAG flows.

For version-specific release notes, see the Graphiti Changelog (see online documentation).

If you are upgrading an existing Graphiti deployment, read both the changelog and the migration section before reusing old data volumes.

Before You Deploy

Graphiti is installed as an extension and requires:

- a deployment-time secret for `NEO4J_PASSWORD`
- persistent storage for Neo4j data
- a reachable LLM or OpenAI-compatible endpoint for message processing

Without an LLM endpoint, Graphiti can still accept requests, but message processing, extraction, and memory/search results will be incomplete or fail.

In App Garden, the install flow prompts for `NEO4J_PASSWORD`. In declarative or Helm-based deployments, supply the same value through your normal secret path (for example a Kubernetes `Secret`, `SealedSecret`, or equivalent env-injection mechanism) instead of hard-coding it in the checked-in manifest.

Runtime Layout

Service	Port	Purpose
`graphiti`	`8000`	FastAPI API for ingest, search, memory, and episode queries
`neo4j`	`7687`	Neo4j Bolt endpoint used by the Graphiti API

Common Graphiti API paths include:

- `GET /healthcheck`
- `POST /messages`
- `POST /search`
- `POST /get-memory`
- `GET /episodes/{group_id}`
- `POST /clear`

Key Configuration

Variable	Purpose
`NEO4J_URI`	Neo4j Bolt URI, usually `bolt://neo4j:7687`
`NEO4J_USER`	Neo4j username, usually `neo4j`
`NEO4J_PASSWORD`	Neo4j password seeded at first boot
`OPENAI_BASE_URL`	Preferred OpenAI-compatible LLM endpoint override
`LLM_BASE_URL`	Graphiti-specific LLM override when you need one
`KAMIWAZA_ENDPOINT`	Fallback model endpoint supplied by Kamiwaza
`EMBEDDING_BASE_URL`	Optional embedding endpoint override
`MODEL_NAME` / `EMBEDDING_MODEL_NAME`	Explicit model selection when auto-selection is not enough

Operational notes:

- In local Compose, Graphiti can fall back between `OPENAI_BASE_URL`, `LLM_BASE_URL`, and `KAMIWAZA_ENDPOINT`.
- In platform deployments, `OPENAI_BASE_URL` is treated as the primary injection point so the platform-managed model endpoint wins by default.
- Neo4j password seeding happens when the data volume is first initialized. Rotating `NEO4J_PASSWORD` later does not rewrite an existing database automatically. See Neo4j authentication changes do not take effect (see online documentation) for the in-place rotation workflow.

Migration from FalkorDB

> **CAUTION**
>
> 0.12.1 replaces FalkorDB with Neo4j. Existing FalkorDB data is not compatible with the new backend, and rollback depends on preserving the pre-upgrade FalkorDB state before you switch the deployment.

Plan upgrades accordingly:

1. Snapshot or otherwise preserve the existing FalkorDB volume before you upgrade.
2. Treat the migration as a storage break.
3. Start Graphiti with a fresh Neo4j data volume.
4. Rebuild the graph from source documents or other canonical inputs.
5. Remove the old FalkorDB volume only after you confirm the new Neo4j-backed service is healthy.

The Graphiti service also uses a different volume naming convention for Neo4j-backed storage to avoid accidental cross-use of FalkorDB data.

Rollback

If you must roll back to the older FalkorDB-backed release:

1. Revert the extension deployment to the last FalkorDB-compatible version.
2. Restore the previous FalkorDB volume or snapshot.
3. Remove or archive the Neo4j volume before retrying the older release.
4. Rebuild graph state from source if no compatible FalkorDB data survives.

Troubleshooting

Search or memory calls fail

Check whether Graphiti can reach a valid LLM endpoint. Missing or invalid model connectivity is the most common reason message ingestion succeeds while retrieval or search fails later.

Neo4j authentication changes do not take effect

Neo4j only seeds credentials on first initialization. If you change `NEO4J_PASSWORD` after the database already exists, either recreate the volume or rotate the password in place inside Neo4j and update the deployment secret/env to match.

For an already-initialized database, rotate it with `cypher-shell` using stdin for the statement body instead of placing both passwords directly on the command line.

Local Docker Compose example:

```
printf 'ALTER CURRENT USER SET PASSWORD FROM "%s" TO "%s";\n' \
  "$OLD_NEO4J_PASSWORD" "$NEW_NEO4J_PASSWORD" \
  | docker compose exec -T \
      -e NEO4J_USERNAME=neo4j \
      -e NEO4J_PASSWORD="$OLD_NEO4J_PASSWORD" \
      neo4j \
      cypher-shell -a bolt://localhost:7687
```

Kubernetes/App Garden example:

```
printf 'ALTER CURRENT USER SET PASSWORD FROM "%s" TO "%s";\n' \
  "$OLD_NEO4J_PASSWORD" "$NEW_NEO4J_PASSWORD" \
  | kubectl exec -i -n <namespace> <neo4j-pod> -- \
      env NEO4J_USERNAME=neo4j NEO4J_PASSWORD="$OLD_NEO4J_PASSWORD" \
      cypher-shell -a bolt://localhost:7687
```

This env-driven pattern matches the shipped Graphiti Neo4j health check, which already invokes `cypher-shell` with `NEO4J_USERNAME` and `NEO4J_PASSWORD` on the pinned Neo4j `v5.26.21` runtime.

If you are seeding a fresh volume before first startup, `neo4j-admin dbms set-initial-password` is the one-time alternative, but it does not replace the in-place rotation flow above for existing databases.

Local testing with custom Graphiti images

Use a `docker-compose.override.yml` file locally to point the `graphiti` service at a custom image without changing the checked-in deployment manifest.

Graphiti Changelog

This document tracks version-specific changes that affect Graphiti deployments and upgrades.

Version 0.12.1

The 0.12.1 line changes Graphiti in ways operators should review before upgrading:

- The backend moved from **FalkorDB** to **Neo4j**.
- The service now runs on pinned prebuilt Graphiti and Neo4j images.
- LLM and embedding endpoint fallback behavior was tightened so the extension aligns cleanly with platform-injected model endpoints.

If you are upgrading an existing Graphiti deployment, also review the migration and rollback guidance in the Graphiti Service Guide (see online documentation).

UNLISTED PAGE

This page is unlisted. Search engines will not index it, and only users having a direct link can access it.

Workroom Manager User Guide (Deprecated)

DEPRECATED

The standalone **Workroom Manager extension** that this guide describes is deprecated and will be removed in a future release. Do not deploy it on new installations that provide the built-in Workrooms page.

Workrooms are built into Kamiwaza. Where your installation provides the built-in page, open **Workrooms** from the navigation sidebar to create, share, and manage workrooms and to launch apps inside a workroom; no extension is required. For the behavior runtimes rely on, see the Workroom Runtime Contract (see online documentation).

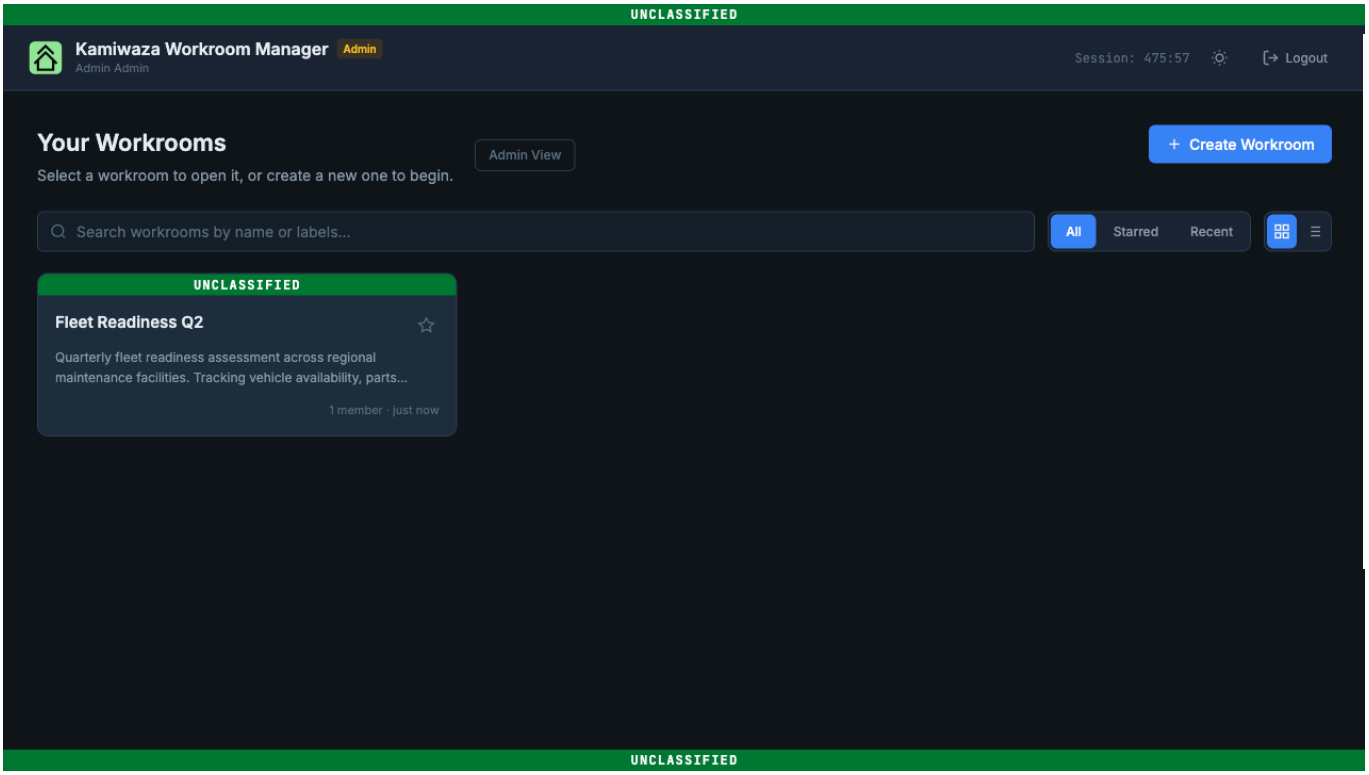
The rest of this page is kept for installations that still provide workrooms through the extension, including earlier releases whose quickstart links here and installations configured to use the extension instead of the built-in page. It describes the extension as it shipped; follow it on those installations, and plan to move to the built-in Workrooms page when your release and configuration support it.

Workroom Manager lets you create focused collaboration spaces inside Kamiwaza. Each **workroom** is an isolated environment with its own classification banner, membership, data, and deployed applications. Use workrooms to organize work by mission, project, or analysis effort, and share them with teammates who need access.

Getting Started

Workroom Manager is available as an extension in your Kamiwaza deployment. Open it from **App Garden** or navigate directly if your administrator has provided a URL.

When you first open Workroom Manager, you see the **Your Workrooms** library. This is your home view — it lists every workroom you own or have been invited to.



From the library you can:

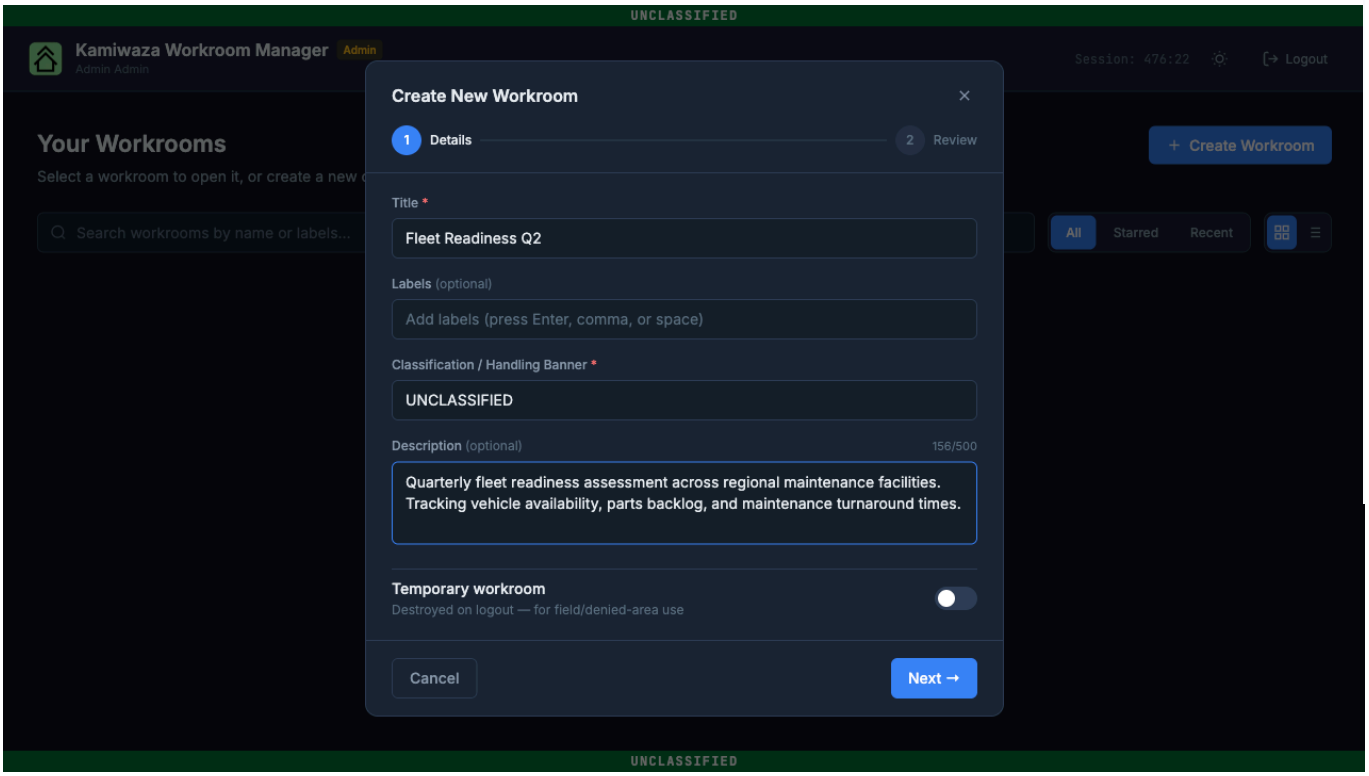
- **Search** workrooms by name or labels
- **Filter** by All, Starred, or Recent
- **Toggle** between grid and list views
- **Create** a new workroom with the **+ Create Workroom** button

Administrators also see an **Admin View** button that shows all workrooms across all users with owner information.

Creating a Workroom

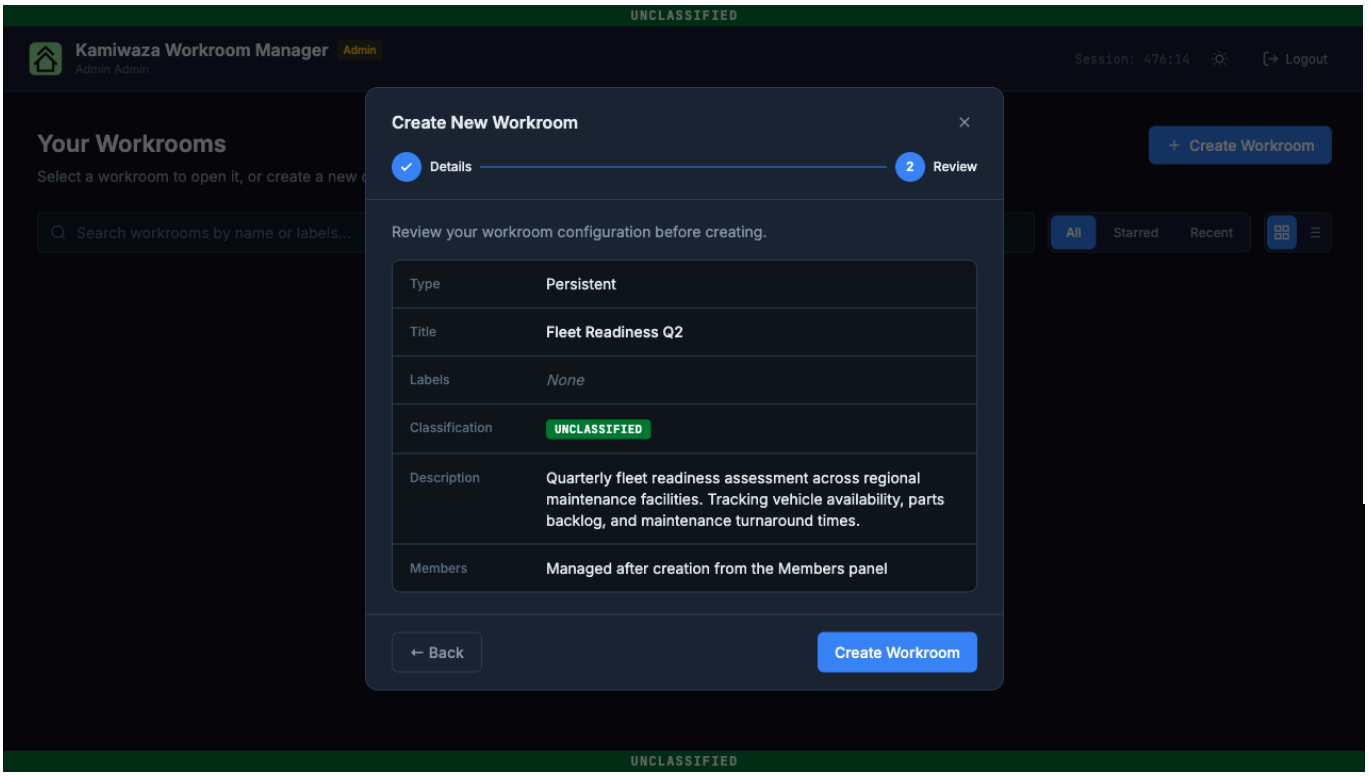
Click **+ Create Workroom** to open the two-step creation wizard.

Step 1: Details



Field	Required	Description
Title	Yes	A short, descriptive name for the mission or analysis effort.
Labels	No	Freeform tags to help organize and search workrooms. Press Enter, comma, or space to add each label.
Classification / Handling Banner	Yes	The handling marking displayed on the workroom (e.g., `UNCLASSIFIED`, `SECRET//NOFORN`).
Description	No	Brief description of the objective or context (up to 500 characters).
Temporary workroom	No	Toggle on for short-lived sessions or field/denied-area use. Temporary workrooms are destroyed on logout. Leave off for persistent workrooms that remain until explicitly deleted.

Step 2: Review



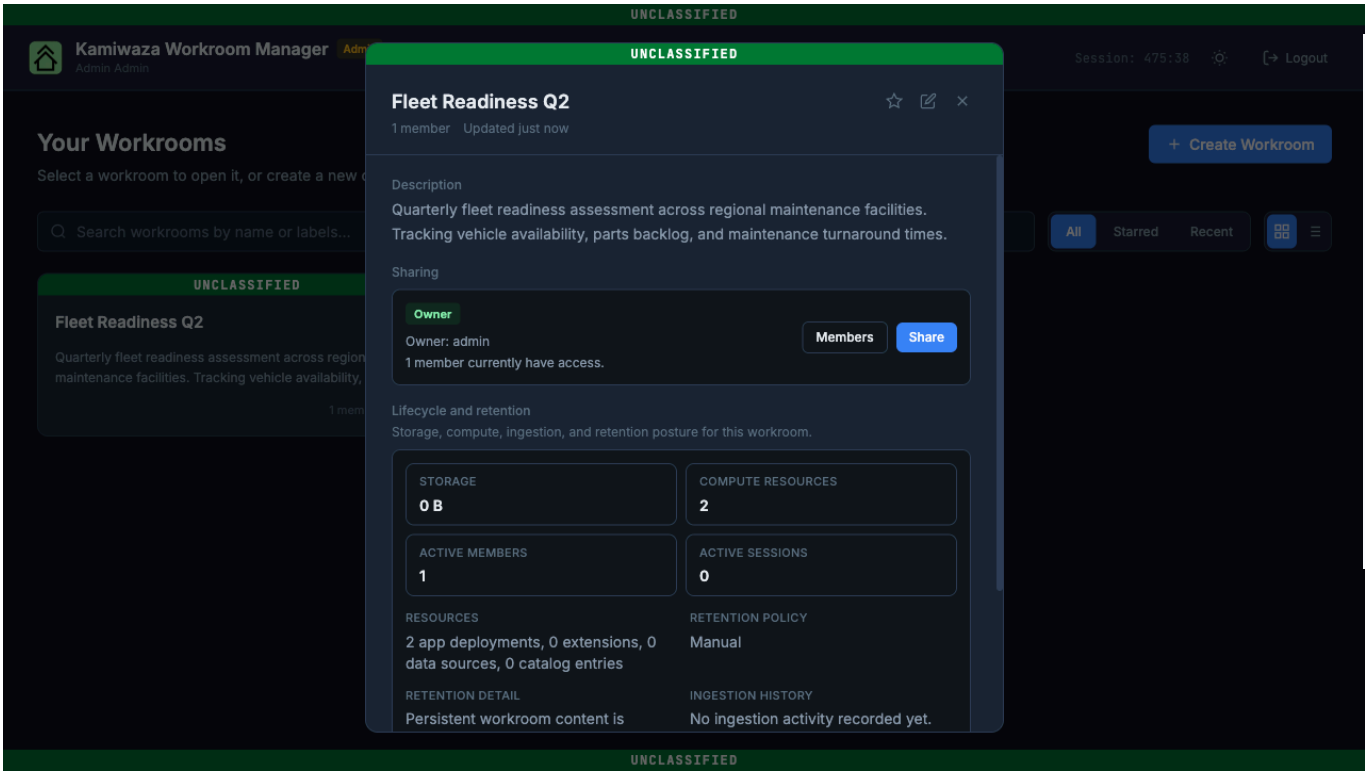
Review the summary table to confirm your settings. The workroom type (Persistent or Temporary) is locked after creation and cannot be changed later.

Click **Create Workroom** to finalize. Your new workroom appears immediately in the library.

> **Note:** Workrooms are created with only you as a member. Add collaborators from the workroom detail view after creation.

Workroom Details

Click any workroom card to open the detail panel.



The detail panel has up to five sections:

Description

The workroom title, classification banner, and description you provided at creation.

Labels

If the workroom has labels, they appear here as tags. This section is only visible when labels have been added. Labels help you organize and search for workrooms as your library grows.

Sharing

Shows the current owner, total member count, and quick-access buttons:

- **Members** — view the full member roster
- **Share** — invite a new member

Lifecycle and Retention

An at-a-glance summary of workroom resource usage and platform signals. These values are intended as operational cues, not as a full accounting report. Exact availability and meaning can vary by deployment and enabled services.

Metric	Description
Storage	Indexed data footprint currently associated with the workroom. This is not a full accounting of every backing volume or runtime artifact.
Compute resources	Number of compute resources allocated
Active members	Members currently using the workroom
Active sessions	Sessions currently bound to this workroom
Resources	Breakdown of app deployments, extensions, data sources, and catalog entries currently attributed to this workroom
Retention policy	The workroom's retention policy as configured by the platform.
Retention detail	Describes when workroom content is retained or removed.
Ingestion history	When data was last ingested into this workroom. Ingestion errors are highlighted if any occurred.

Apps

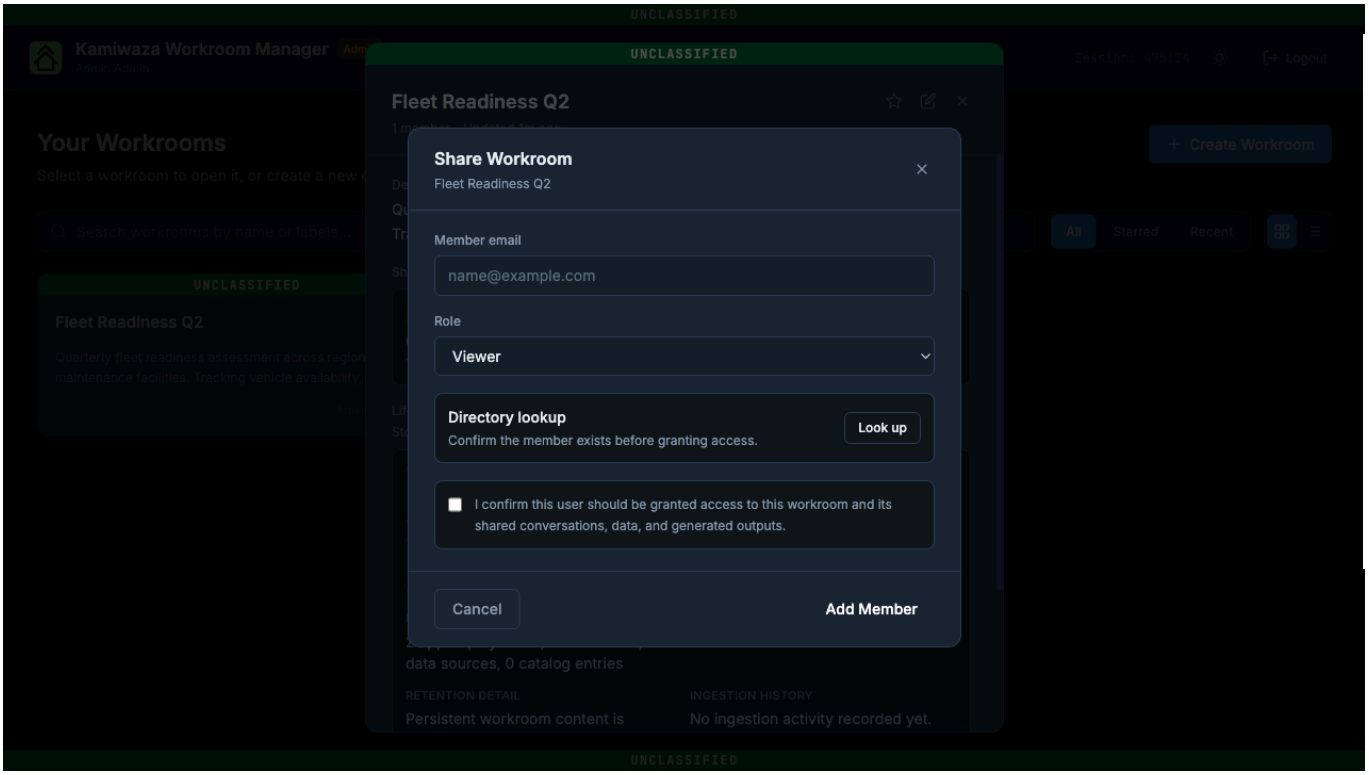
Lists applications available in the workroom. Click **Launch** to start an application (such as Kaizen) in this workroom's context, or **Open** if it is already running.

When Kaizen is launched from a shared non-global workroom, authorized members see the same workroom-scoped agents, conversations, uploaded data, and generated outputs according to role. The **Global Workroom** remains a special system workspace and does not automatically expose personal Kaizen history across users.

For details on using Kaizen, see the Kaizen User Guide (see online documentation).

Sharing a Workroom

From the workroom detail panel, click **Share** to invite a collaborator.



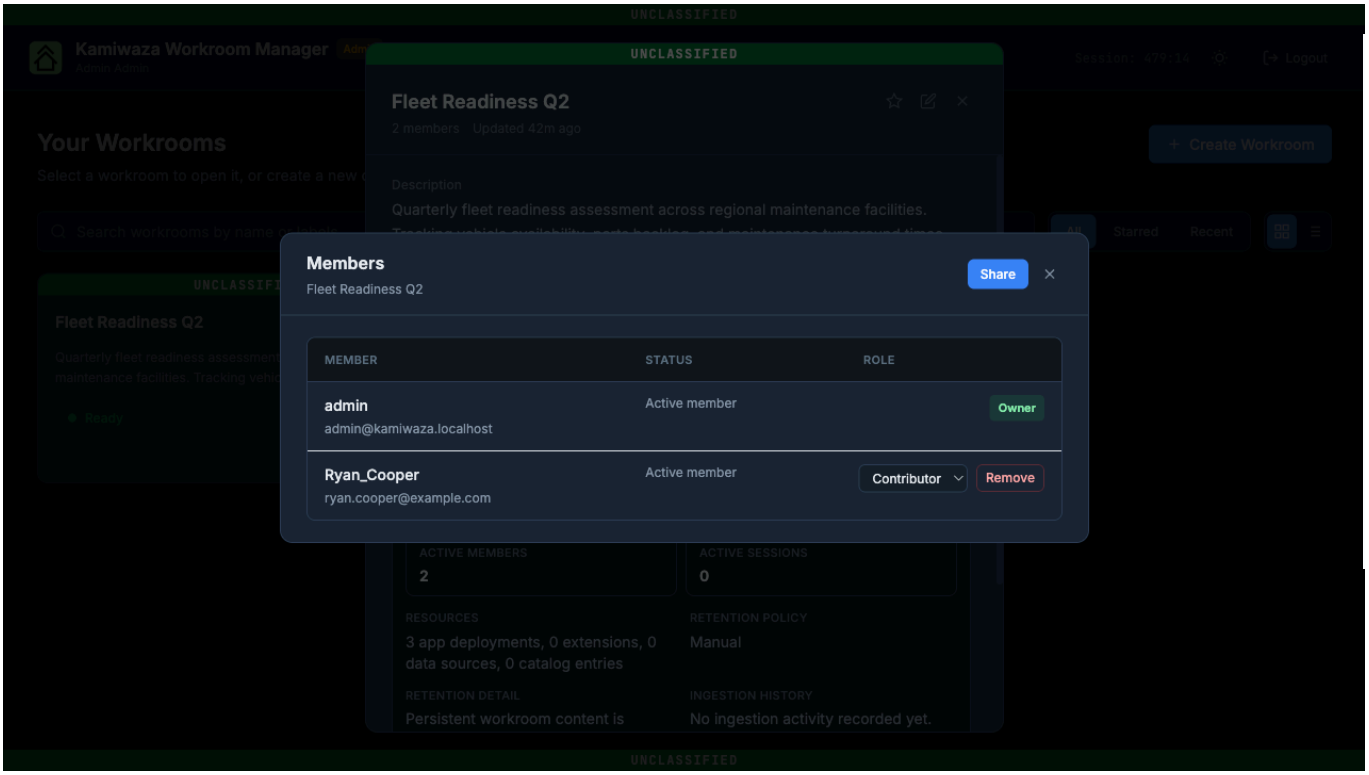
Adding a member

1. Enter the member's **email address**.
2. Select a **Role**:
   - **Viewer** — read-only access to the workroom and its contents.
   - **Contributor** — can participate in the workroom and its applications.
3. Click **Look up** to verify the member exists in the directory. This confirms the user account is valid before granting access.
4. Check the **attestation checkbox**: *"I confirm this user should be granted access to this workroom and its shared conversations, data, and generated outputs."*
5. Click **Add Member**.

> The attestation step ensures that access is granted intentionally. You must confirm the checkbox before the Add Member button becomes active.

Managing members

Click **Members** from the workroom detail panel to view the current roster.



The members table shows each member's name, email, status, and role.

Owners can use this panel to change non-owner roles, remove members, and transfer primary ownership to another member when responsibility for the workroom needs to change hands.

Workroom roles

Role	Capabilities
Owner	Full control — update settings, manage members, transfer ownership, share the workroom, and delete the workroom. In Kaizen, Owners can also continue shared workroom conversations and create or update shared agents. Assigned to the workroom creator.
Contributor	Participate in the workroom, use deployed applications, and collaborate with other members. In Kaizen, Contributors can create shared agents and conversations visible to other workroom members.
Viewer	Read-only access to the workroom and its contents, including shared Kaizen history and generated outputs.

Owners can change a member's role between Contributor and Viewer, remove a member entirely, or transfer ownership from the Members panel. If you remove a member who has an active session, the app may ask for confirmation before proceeding.

In shared non-global workrooms, membership is the sharing boundary — you do not need to configure per-conversation or per-agent sharing inside Kaizen for another authorized member to see shared workroom history. Credential sharing remains separate from content sharing; workroom membership does not automatically grant reuse of another member's personal credentials unless explicitly shared through a supported credential path.

Collaboration Status

Workroom Manager surfaces collaboration status so you can tell whether a workroom is currently in use before making changes.

You may see:

- An **active-member count** in the workroom library card or detail view
- **Per-member active session counts** in the membership view
- **Warnings** when you try to remove a member who still has an active session

These indicators help owners understand whether a workroom is currently in use before they change access or remove participants.

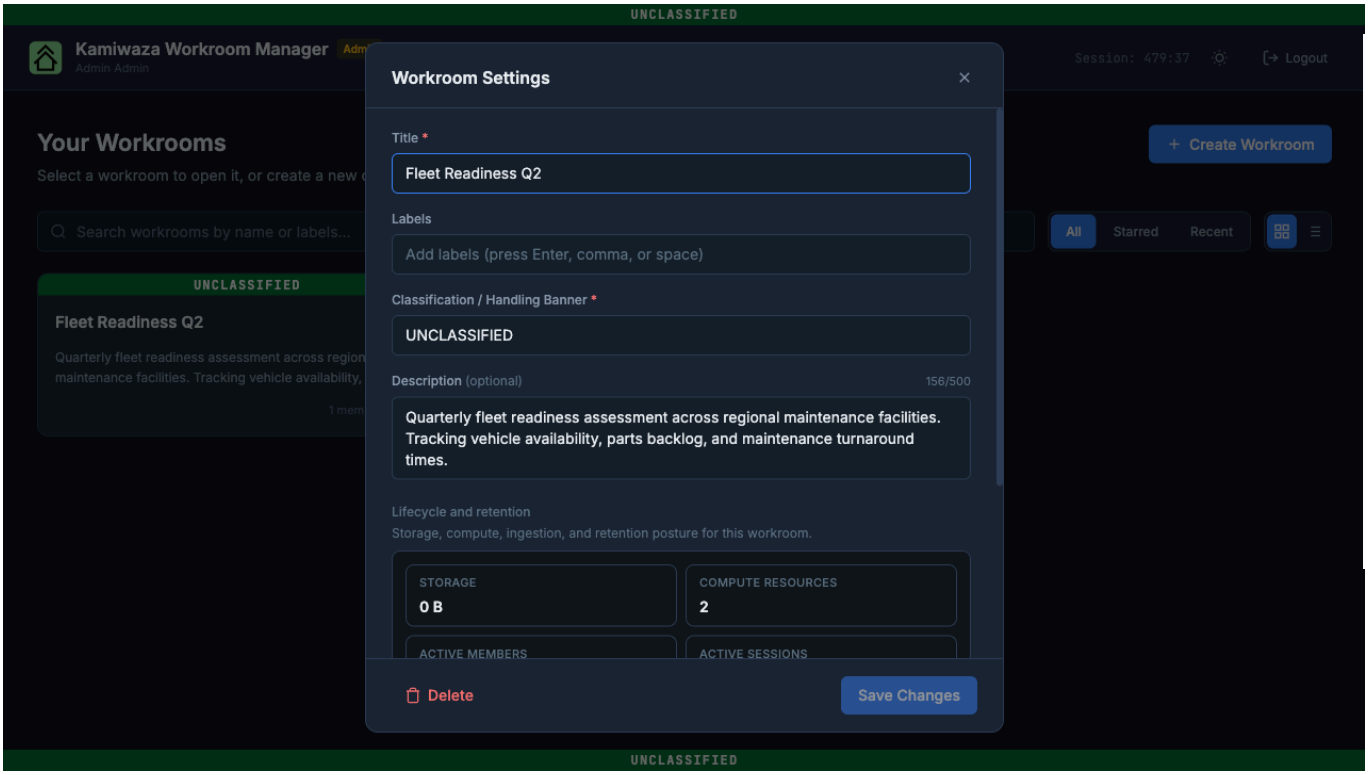
For the public runtime, membership, and collaboration-stream contract behind these behaviors, see the main-docs Workroom Runtime Contract (see online documentation).

Starring Workrooms

Click the **star icon** on any workroom card or in the detail panel header to mark it as a favorite. Starred workrooms appear when you select the **Starred** filter in the library, giving you quick access to the workrooms you use most often.

Editing a Workroom

Click the **edit icon** (pencil) in the workroom detail panel header to open **Workroom Settings**.



You can edit:

- Title
- Labels
- Classification / Handling Banner
- Description

The settings dialog also shows the lifecycle and retention metrics for the workroom. The workroom type (Persistent or Temporary) cannot be changed after creation.

Click **Save Changes** to apply your updates.

Deleting a Workroom

From the **Workroom Settings** dialog, click the red **Delete** button at the bottom left. A confirmation dialog appears:

> *"Are you sure you want to delete [workroom name]? This action cannot be undone."*

Click **Delete Workroom** to confirm. Deletion removes the workroom and all associated data permanently.

Administrators can also delete any workroom from Admin View by clicking the workroom card and selecting the delete icon in the detail panel header.

Lifecycle Notes

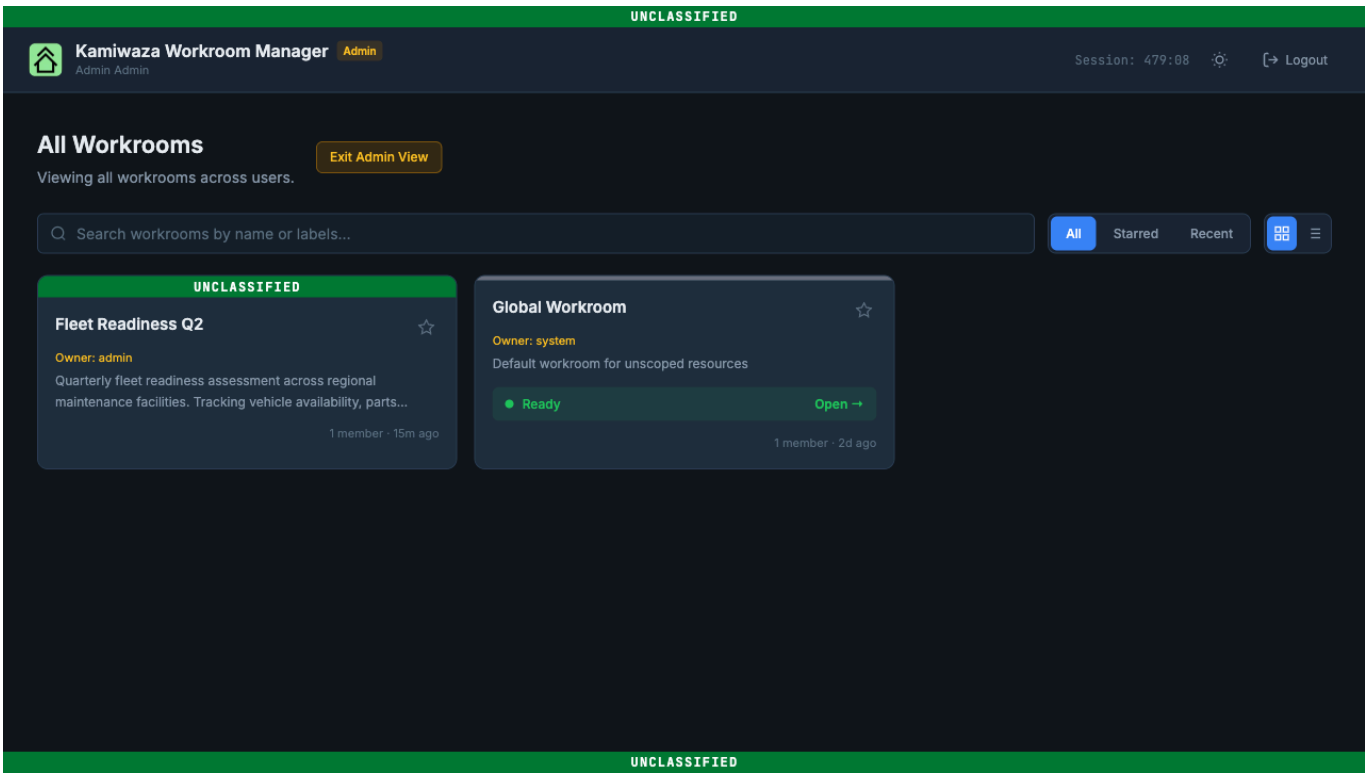
Keep these workroom behaviors in mind:

- **Temporary workrooms** are designed for short-lived sessions and are destroyed on logout.
- **Persistent workrooms** remain in the library until explicitly removed by an owner.
- Lifecycle states such as **archived** or **deleting** can temporarily disable editing and member changes while the workroom is read-only.

Archive, restore, lifecycle-summary, and export-bundle capabilities exist in the underlying Kamiwaza workroom APIs. As these become available in the Workroom Manager interface, this guide will be updated.

Admin View

Platform administrators see an **Admin View** button on the main library page. Toggling Admin View shows:



- **All workrooms** across all users, not just your own
- **Owner** information for each workroom
- The **Global Workroom** (the default system workroom for unscoped resources)
- **Delete** capability for any workroom except the Global Workroom, which cannot be deleted

Click **Exit Admin View** to return to your personal workroom library.

> **Note:** Admin View is only visible to users with administrator privileges.

Tips

- **Use descriptive titles** — titles like "Fleet Readiness Q2" are easier to find than generic names.
- **Add labels** — labels make workrooms searchable as your library grows.
- **Check membership before sharing sensitive data** — review the Members panel to confirm who has access.
- **Use temporary workrooms for short-lived tasks** — they clean up automatically on logout.

Troubleshooting

I cannot launch Kaizen

Check the deployment status in the workroom detail panel under Apps. If the status does not become Ready, review the Workroom Manager or Kaizen deployment logs. For platform-level logs, see Observability (see online documentation).

I cannot add or remove members

Only workroom owners can change membership. If the workroom is in a read-only lifecycle state (e.g., archived or deleting), member changes may be blocked until the state clears.

I cannot see the full member roster

Non-owners can view the member roster but cannot change roles or remove members. If the roster is not loading, check your network connection and ensure the workroom is in an active state.

A workroom delete or edit failed

Retry the operation once. If the failure persists, check the Workroom Manager deployment logs for errors near the time of the action. Include the workroom name and any visible error message if you escalate the issue.

Learn More

- Kaizen User Guide (see online documentation)
- Skills Library Guide (see online documentation)
- Quickstart (see online documentation)
- Administrator Guide (see online documentation)
- Observability (see online documentation)