# Kamiwaza AI Platform

Platform Documentation

Version 0.6.0

## Table of Contents

29. SDK - Catalog
30. SDK - Cluster
31. SDK - Embedding
32. SDK - Ingestion
33. SDK - Lab
34. SDK - Models
35. SDK - Retrieval
36. SDK - Serving
37. SDK - Vectordb

# Welcome to Kamiwaza AI Docs

Welcome to the official documentation for the Kamiwaza AI Platform, the enterprise 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 our **Installation Guide** to set up your environment.

## Looking for Inspiration?

Check out our **Use Cases** section to see practical implementations and get inspired by what's possible with Kamiwaza.

## Explore the Documentation

### 🚀 Installation Guide

Set up your local environment and get Kamiwaza running in minutes.

Start the Installation →

### ⚡ Quickstart

Get up and running with a real application from the App Garden.

Launch an App →

### 🤖 Models

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

Manage Models →

### 🪴 App Garden

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

Explore Apps →

### 💾 Distributed Data Engine

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

Learn About Data →

### 📦 SDK

Interact with the Kamiwaza platform programmatically using our Python SDK.

Explore the SDK →

### 🏛️ Platform Architecture

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

View the Architecture →

### 💡 Use Cases

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

Explore Use Cases →

### 📚 Other Topics

Dive into cluster management, activity logging, administration, and more.

See Other Topics →

## Need Help?

If you have questions or run into issues, we're here to help:

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

We're committed to making your experience with Kamiwaza as smooth as possible.

# Installing Kamiwaza

## Before You Begin

**Please review the System Requirements before proceeding with installation.** This document covers:

- Supported operating systems and versions
- Hardware requirements (CPU, RAM, storage)
- Required system packages and dependencies
- Network and storage configuration
- GPU support requirements

## Installation Workflows

### Linux

#### Ubuntu .deb Package Installation (for Ubuntu 24.04 Noble)

1. Add Kamiwaza repository to APT sources

```
echo "deb [signed-by=/usr/share/keyrings/kamiwaza-archive-keyring.gpg]
https://packages.kamiwaza.ai/ubuntu/ noble main" | sudo tee
/etc/apt/sources.list.d/kamiwaza.list
```

2. Import and install Kamiwaza GPG signing key

```
curl -fsSL https://packages.kamiwaza.ai/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/kamiwaza-archive-keyring.gpg
```

3. Update package database and install Kamiwaza

```
sudo apt update
sudo apt upgrade
sudo apt install kamiwaza
```

4. Verify service starts (see Quickstart)

### RHEL .rpm Package Installation (for RHEL 9)

For offline/air-gapped RHEL installations, see the comprehensive Red Hat Offline Installation Guide.

### Other Linux Distros via Tarball

1. Follow the consolidated guide: Linux/macOS tarball installation
2. Ensure Docker Engine (with Compose v2), Python 3.10, and Node.js 22 are available (installer may configure as needed)
3. Run `install.sh --community`
4. Access via browser at `https://localhost`

## Community Edition on macOS

*Only Community Edition is supported on macOS.*

1. Follow the consolidated guide: Linux/macOS tarball installation
2. Ensure Docker Engine (with Compose v2), Python 3.10, and Node.js 22 are available (installer may configure as needed)
3. Run `install.sh --community`
4. Access via browser at `https://localhost`

## Community Edition on Windows

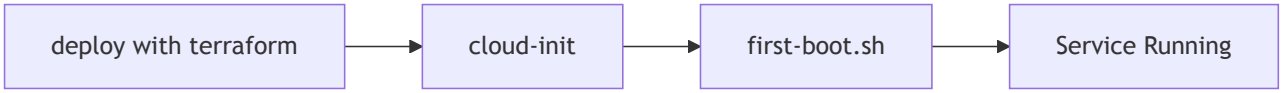
Use the MSI installer for a streamlined WSL2-based setup. See the Windows Installation Guide for prerequisites, GPU support, and step-by-step instructions.

Steps:

1. Download: `KamiwazaInstaller-[version]-[arch].msi`
2. Install: Run the MSI (reboot when prompted)
3. Launch: Start Menu → "Kamiwaza Start"

### Enterprise Edition Deployment

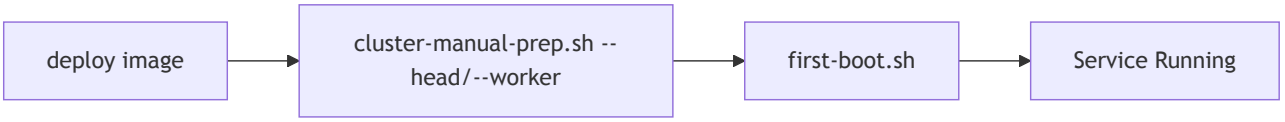
#### A. Terraform Deployment (Recommended)




Key Points:

- Terraform handles complete cluster setup
- cloud-init automatically runs first-boot.sh
- Service starts automatically via systemd

#### B. Manual Cluster Deployment




Key Points:

- Requires manual cluster setup via cluster-manual-prep.sh
- Must specify correct role (`--head` or `--worker --head-ip=<IP>`)
- Service starts automatically via systemd

## Updating Kamiwaza

### Windows

- Download new MSI installer and run to update existing installation
- Restart if prompted for GPU changes

### Linux/macOS

- Download new package
- Run installation script again
- Service will restart automatically

## Uninstallation

### Windows

- Windows Settings → Add or Remove Programs -> (three dots on side) Uninstall

### Linux/macOS

- Remove package via package manager
- Clean up any remaining configuration files

# System Requirements

## Base System Requirements

### Supported Operating Systems & Architecture

- **Linux**:
  - Ubuntu 24.04 and 22.04 LTS via .deb package installation (x64/amd64 architecture only)
  - Redhat Enterprise Linux (RHEL) 9
- **Windows**: 11 (x64 architecture) via WSL with MSI installer
- **macOS**: 12.0 or later, Apple Silicon (ARM64) only (community edition only)

### CPU Requirements

- **Architecture**:
  - Linux: x64/amd64 (64-bit)
  - macOS: ARM64 (Apple Silicon) only
  - Windows: x64 (64-bit)
- **Minimum Cores**: 8+ cores
- **Recommended Cores**: 16+ cores for CPU-based inference workloads

### Core Software Requirements

- **Python**: Python 3.10 for tarball installations; Python 3.12 for .deb/.msi installations
- **Docker**: Docker Engine with Compose v2
- **Node.js**: 22.x (installed via NVM during setup)
- **Browser**: Chrome Version 141+ (tested and recommended)
- **GPU Support**: NVIDIA GPU with compute capability 7.0+ (Linux only) or NVIDIA RTX/Intel Arc (Windows via WSL)

## Memory Requirements

### System RAM

- **Minimum**: 16GB RAM
- **Recommended**: 32GB+ RAM for CPU-based inference workloads
- **GPU Workloads**: 16GB+ system RAM (32GB+ recommended)

### GPU Memory (vRAM)

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

### Windows (WSL-based) Specific

- **Minimum**: 16GB RAM
- **Recommended**: 32GB+ RAM
- **Memory Allocation**: 50-75% of system RAM dedicated to Kamiwaza during installation

## Storage Requirements

### Storage Performance

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

### Storage Capacity

#### Linux/macOS

- **Minimum**: 100GB free disk space
- **Recommended**: 200GB+ free disk space
- **Enterprise Edition**: Additional space for /opt/kamiwaza persistence

#### Windows

- **Minimum**: 100GB free disk space
- **Recommended**: 200GB+ free space on SSD

- **WSL**: Automatically manages Ubuntu 24.04 installation space

📋 **For detailed Windows storage and configuration requirements, see the Windows Installation Guide.**

## 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), MLX (Apple Silicon)
- **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)
- **Storage:** 200GB NVMe SSD (100GB minimum)
- **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:** 1-2TB NVMe SSD
- **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
- **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 Enterprise Edition production clusters, avoid non-homogeneous hardware (e.g., GPU-less head nodes). Each node participates in data plane duties (Traefik 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 | 200GB gp3 |
| **Tier 1: With GPU** | `g5.xlarge` | 4 | 16GB | 1x A10G (24GB) | 200GB gp3 |
| **Tier 1: Alternative** | `g5.2xlarge` | 8 | 32GB | 1x A10G (24GB) | 200GB 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:**

- 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 | 200GB SSD |
| **Tier 1: With GPU** | `n1-standard-8` + `1x T4` | 8 | 30GB | 1x T4 (16GB) | 200GB SSD |
| **Tier 1: Alternative** | `g2-standard-8` + `1x L4` | 8 | 32GB | 1x L4 (24GB) | 200GB 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:**

- 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 | 200GB Premium SSD |
| **Tier 1: With GPU** | `Standard_NC4as_T4_v3` | 4 | 28GB | 1x T4 (16GB) | 200GB Premium SSD |
| **Tier 1: Alternative** | `Standard_NC6s_v3` | 6 | 112GB | 1x V100 (16GB) | 200GB 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:**

- 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.

### Windows-Specific Prerequisites

- Windows Subsystem for Linux (WSL) installed and enabled
- Administrator access required for initial setup
- Windows Terminal (recommended for optimal WSL experience)

## Dependencies & Components

### Required System Packages

See platform-specific installation instructions

### NVIDIA Components (Linux GPU Support)

- NVIDIA Driver (550-server recommended)
- NVIDIA Container Toolkit
- nvidia-docker2

### Windows Components (Automated via MSI Installer)

- Windows Subsystem for Linux (WSL 2)
- Ubuntu 24.04 LTS (automatically downloaded and configured)
- Docker Engine (configured within WSL)
- GPU drivers and runtime (automatically detected and configured)
- Node.js 22 (via NVM within WSL environment)

### Docker Configuration Requirements

- Docker Engine with Compose v2
- User must be in docker group
- Swarm mode (Enterprise Edition)

- Docker data root configuration (configurable)

### Required Directory Structure

#### Enterprise Edition

Note this is created by the installer and present in cloud marketplace images.

```
/etc/kamiwaza/
├── config/
├── ssl/      # Cluster certificates
└── swarm/    # Swarm tokens

/opt/kamiwaza/
├── containers/  # Docker root (configurable)
├── logs/
├── nvm/        # Node Version Manager
└── runtime/    # Runtime files
```

#### Community Edition

We recommend `${HOME}/kamiwaza` or something similar for `KAMIWAZA_ROOT`.

```
$KAMIWAZA_ROOT/
├── env.sh
├── runtime/
└── logs/
```

## 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

## Required Kernel Modules (Enterprise Edition Linux Only)

Required modules for Swarm container networking:

- overlay
- br_netfilter

## System Network Parameters (Enterprise Edition Linux Only)

These will be set by the installer.

```
# Required sysctl settings for Swarm networking
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
```

## Community Edition Networking

- Uses standard Docker bridge networks

- No special kernel modules or sysctl settings required
- Simplified single-node networking configuration

## Detailed Storage Requirements

### Capacity Planning

| Component | Minimum | Recommended | Notes |
|---|---|---|---|
| **Operating System** | 20GB | 50GB | Ubuntu/RHEL base + dependencies |
| **Kamiwaza Platform** | 50GB | 50GB | Python environment, Ray, services |
| **Model Storage** | 50GB | 500GB+ | Depends on number and size of models |
| **Database** | 10GB | 50GB | CockroachDB 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** | **170GB** | **900GB+** | |

### Storage Performance Requirements

#### Local Storage (Single Node)

**Storage Type:**

- **Minimum:** SATA SSD (500 MB/s sequential read)
- **Recommended:** NVMe SSD (2000+ MB/s sequential read)
- **Note:** HDD: 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

### 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 |

## Storage Configuration by Edition

### Enterprise Edition Requirements

- Primary mountpoint for persistent storage (/opt/kamiwaza)
- Scratch/temporary storage (auto-configured)
- For Azure: Additional managed disk for persistence
- Shared storage for multi-node clusters (see Shared Storage Options above)

### Community Edition

- Local filesystem storage
- Configurable paths via environment variables
- Single-node storage only (no shared storage required)

## Special Considerations

### Apple Silicon (M-Series)

**MLX Engine Support:**

- Kamiwaza supports Apple Silicon via the MLX inference engine
- Unified memory architecture (shared CPU/GPU RAM)
- Excellent performance for models up to 13B parameters; reasonable performance for larger models when context is appropriately restricted and RAM is available.
- All M-series chips work in approximately the same way, but newer chips (e.g., M4) offer substantially higher performance than older versions
- Ultra chips (Mac Studio/Mac Pro models) typically offer 50-80% more performance than Pro versions

**Notes:**

- No tensor parallelism support (single chip only)
- Not for production use; like-for-like API, UI, capabilities.
- Community edition only; single node only (Enterprise edition not available on macOS)

## Important Notes

- **System Impact**: Network and kernel configurations can affect other services
- **Security**: Certificate generation and management for cluster communications
- **GPU Support**: Available on Linux (NVIDIA GPUs) and Windows (NVIDIA RTX, Intel Arc via WSL)
- **Storage**: Enterprise Edition requires specific storage configuration
- **Network**: Enterprise Edition requires specific network ports for cluster communication
- **Docker**: Custom Docker root configuration may affect other containers
- **Windows Edition**: Requires WSL 2 and will create a dedicated Ubuntu 24.04 instance
- **Administrator Access**: Windows installation requires administrator privileges for initial setup

## Additional Considerations

### Network Ports

#### Linux/macOS Enterprise Edition

- 443/tcp: HTTPS primary access
- 51100-51199/tcp: Deployment ports for model instances (will also be used for 'App Garden' in the future)

#### Windows Edition

- 443/tcp: HTTPS primary access (via WSL)
- 61100-61299/tcp: Reserved ports for Windows installation

### Version Compatibility

- Docker Engine: 20.10 or later
- NVIDIA Driver: 450.80.02 or later
- ETCD: 3.5 or later
- Node.js: 22.x (installed automatically)

# Community Edition Installation on Linux and macOS

This guide covers installing Kamiwaza Community Edition on Linux and macOS using the pre-built tarball bundles.

## Before you start

- Review the System Requirements
- Ensure you have administrator/sudo privileges
- Recommended: Latest Docker Desktop (macOS) or Docker Engine (Linux)

## macOS (Sequoia 15+)

### 1) Install Homebrew and core tools

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew update
brew install pyenv pyenv-virtualenv docker cairo gobject-introspection jq cfssl
etcd cmake
brew install cockroachdb/tap/cockroach
```

### 2) Install Docker Desktop

```
brew install --cask docker
open -a Docker
# optional if Docker created files as root
sudo chown -R "$(whoami)":staff ~/.docker || true
```

### 3) Configure Python 3.10 with pyenv

```
echo 'export PATH="$HOME/.pyenv/bin:$PATH"' >> ~/.zshrc
echo 'eval "$(pyenv init -)"' >> ~/.zshrc
echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.zshrc
source ~/.zshrc
pyenv install 3.10
pyenv local 3.10
```

### 4) Install Node.js 22 with NVM

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
export NVM_DIR="${XDG_CONFIG_HOME:-$HOME/.nvm}"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
nvm install 22
```

### 5) Download and install Kamiwaza (tarball)

```
mkdir -p ~/kamiwaza && cd ~/kamiwaza
# Example for 0.5.1 (replace with the latest available version if needed)
curl -L -O https://github.com/kamiwaza-ai/kamiwaza-community-
edition/raw/main/kamiwaza-community-0.5.1-OSX.tar.gz
tar -xvf kamiwaza-community-0.5.1-OSX.tar.gz
bash install.sh --community
```

## Linux (Ubuntu 22.04 and 24.04 LTS)

### 1) [For Ubuntu 24.04 only] Install Python 3.10

Kamiwaza CE requires Python 3.10. These commands will install Python 3.10 on Ubuntu 24.04.

```
sudo apt update
sudo apt install software-properties-common -y
```

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update && sudo apt upgrade -y
```

```
sudo apt install -y python3.10
```

```
sudo ln -sf /usr/bin/python3.10 /usr/local/bin/python
```

## 2) System update and core packages

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y python3.10 python3.10-dev libpython3.10-dev python3.10-venv
golang-cfssl python-is-python3 etcd-client net-tools curl jq libcairo2-dev
libgirepository1.0-dev
```

## 3) Node.js 22 with NVM

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
export NVM_DIR="${XDG_CONFIG_HOME:-$HOME/.nvm}"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
nvm install 22
```

### 4) Docker Engine + Compose v2

```
sudo apt install -y apt-transport-https ca-certificates curl software-properties-
common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
sudo mkdir -p /usr/local/lib/docker/cli-plugins
sudo curl -SL
"https://github.com/docker/compose/releases/download/v2.39.1/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/lib/docker/cli-plugins/docker-
compose
sudo chmod +x /usr/local/lib/docker/cli-plugins/docker-compose
sudo usermod -aG docker $USER
sudo chown $USER:$USER /var/run/docker.sock
# Log out and back in (or reboot) so the docker group membership takes effect
```

### 5) Install CockroachDB and additional dependencies

```
wget -qO- https://binaries.cockroachdb.com/cockroach-v23.2.12.linux-amd64.tgz |
tar xvz
sudo cp cockroach-v23.2.12.linux-amd64/cockroach /usr/local/bin
sudo apt install -y libcairo2-dev libgirepository1.0-dev
```

### 6) (Optional) NVIDIA GPU support

Use this section if all of the following are true:

- You are on Ubuntu 22.04 or 24.04 (bare metal or a VM with GPU passthrough)
- The host has an NVIDIA GPU and you want GPU acceleration
- You are not on macOS (macOS does not support NVIDIA GPUs)

If you are installing on an Ubuntu 22.04 or 24.04 instance with an NVIDIA GPU where `nvidia-smi` doesn't work, you likely need to do this. However, many cloud-provided images come with NVIDIA drivers pre-installed.

Install the recommended NVIDIA driver, then the NVIDIA Container Toolkit, and configure Docker:

```
# 1) Install the recommended NVIDIA driver
## If 'ubuntu-drivers' is missing, install it first:
##   sudo apt update && sudo apt install -y ubuntu-drivers-common
sudo apt update
sudo ubuntu-drivers autoinstall
```

Perform system reboot:

```
sudo reboot
```

After the reboot, install the container toolkit and configure Docker:

```
# 2) Install NVIDIA Container Toolkit repository and package
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | \
  sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-
container-toolkit.list | \
  sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-
toolkit-keyring.gpg] https://#g' | \
  sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list > /dev/null
sudo apt update
sudo apt install -y nvidia-container-toolkit

# 3) Configure Docker to use the NVIDIA runtime and restart Docker
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker

# 4) Test GPU access from Docker (should print nvidia-smi output and exit)
docker run --rm --gpus all nvidia/cuda:12.4.1-runtime-ubuntu22.04 nvidia-smi
```

Verify the driver is installed with:

```
nvidia-smi
```

Notes:

- If Secure Boot is enabled, you may be prompted to enroll MOK during driver installation.

- On some servers, you may prefer `nvidia-driver-550-server`. If you need a specific version: `sudo apt install -y nvidia-driver-550-server`.
- The `nvidia-docker2` meta-package is no longer required; use `nvidia-container-toolkit` with `nvidia-ctk` instead.

### 7) Download and install Kamiwaza (tarball)

```
mkdir -p ~/kamiwaza && cd ~/kamiwaza
# Example for 0.5.1 (replace with the latest available version if needed)
wget https://github.com/kamiwaza-ai/kamiwaza-community-edition/raw/main/kamiwaza-
community-0.5.1-UbuntuLinux.tar.gz
tar -xvf kamiwaza-community-0.5.1-UbuntuLinux.tar.gz
bash install.sh --community
```

## Start the platform

After installation completes:

```
# Community Edition
bash startup/kamiwazad.sh start
```

Access the web console at `https://localhost`

- Default Username: `admin`
- Default Password: `kamiwaza`

## Troubleshooting

- Docker permissions: ensure your user is in the `docker` group (Linux) and re-login/reboot.
- Python version: Kamiwaza requires Python 3.10. If you used 3.11+, reinstall 3.10 and rerun the installer.
- GPU: For Linux NVIDIA issues, validate `nvidia-smi` works inside Docker as shown above. For Windows GPU setup, see Windows GPU Setup Guide.

## Notes

- Replace example tarball URLs with the latest version as needed.
- The installer sets up virtual environments, required packages, and initial configuration automatically.

# Windows Installation Guide

## System Requirements

### Minimum Requirements

- **Operating System**: Windows 11
- **Memory**: 8GB RAM minimum (16GB+ recommended)
- **GPU**: NVIDIA modern GPUs and Intel Arc Supported (requires drivers)
- **Storage**: 20GB free disk space
- **Architecture**: x64 (64-bit) processor
- **Administrator Access**: The installer will request permission when needed

### Recommended for Optimal Performance

- **Memory**: 32GB+ RAM for large workloads
- **GPU**: NVIDIA GeForce RTX series or Intel Arc GPU (for hardware acceleration)
- **Storage**: SSD with 50GB+ free space

## Prerequisites Setup

### Step 1: Enable WSL (Windows Subsystem for Linux)

If WSL is not already installed on your system:

1. Open PowerShell as Administrator
   - Right-click Start button → "Windows PowerShell (Admin)"
2. Install WSL

   ```
   wsl --install
   ```

3. Restart your computer when prompted
4. Verify WSL installation

```
wsl --version
```

### Step 2: Verify GPU Access (If Applicable)

**Note**: GPU verification will be performed automatically during installation. The installer will detect and configure GPU access for supported hardware.

**Supported GPUs:**

- NVIDIA GeForce RTX series (30, 40 and 50 series)
- Intel Arc GPUs

**Note**: Kamiwaza currently supports only NVIDIA GPUs and Intel Arc GPUs for hardware acceleration. For Intel Arc GPU setup instructions, please refer to the separate Intel Arc WSL GPU virtualization documentation.

### Step 3: Install Windows Terminal (Optional but Recommended)

Download from Microsoft Store or GitHub releases

## Download and Installation

### Step 1: Download Kamiwaza Installer

Contact your Kamiwaza representative to obtain the installer download link or file:

- **File**: https://packages.kamiwaza.ai/win/kamiwaza_installer_0.5.1_x86_64.msi (https://packages.kamiwaza.ai/win/kamiwaza_installer_0.5.1_x86_64.msi)
- **Size**: Approximately 30-40MB

### Step 2: Run the Installer

1. Locate the downloaded MSI file in your Downloads folder
2. Double-click to run the installer
3. When prompted by Windows User Account Control, click "Yes" to allow the installer to make changes to your device
4. Follow the installation wizard

**Configuration Options:**

- **Email Address**: Your registered email address
- **License Key**: Provided by Kamiwaza support
- **Installation Mode**:
  - **Lite** - Basic installation (recommended for most users)
  - **Full** - Complete installation with all features
- **Dedicated Memory**: Select RAM allocation for Kamiwaza
  - **Recommended**: 50%-75% of total system RAM
  - **Example**: 16GB system → Select 12GB allocation

### Step 3: Installation Process

The installer will automatically:

- Download and install Ubuntu 24.04 WSL distribution (if not present)
- Reserve network ports (61100-61299)
- Detect GPU hardware (NVIDIA RTX, Intel Arc only)
- Configure WSL environment with optimized settings
- Install Kamiwaza platform in dedicated WSL instance
- Setup GPU acceleration (if compatible hardware detected)

**Expected Installation Time:**

- **Standard Installation**: 15-30 minutes
- **First-time WSL Setup**: Add 10-15 minutes
- **Large Package Downloads**: May take longer on slower connections

### Step 4: GPU Driver Restart

If GPU acceleration was configured, you'll be prompted to restart your device. It is recommended to restart immediately to ensure proper GPU driver initialization.

## Access Your Installation

### Option 1: System Tray Access (Primary Method)

After installation, Kamiwaza will automatically launch and appear in your system tray. Right-click the Kamiwaza system tray icon to access the following options:

- **Show Kamiwaza Manager** - Open the main management interface
- **Kamiwaza Status** - Check current platform status
- **Start Kamiwaza** - Start the platform if stopped
- **Stop Kamiwaza** - Stop the running platform
- **Open Kamiwaza** - Launch the web interface
- **Exit** - Close the system tray application

### Option 2: WSL Command Line Access

Access the Kamiwaza WSL environment and start the platform:

```
wsl -d kamiwaza
kamiwaza start
```

### Option 3: Start Menu Shortcuts

After installation, find these shortcuts in Start Menu → "Kamiwaza":

- **Install Kamiwaza** - Initial setup and installation
- **Start Platform** - Launch Kamiwaza platform
- **Cleanup WSL** - Complete removal tool

### Option 4: Direct Browser Access

Once running, access Kamiwaza at:

- **URL**: `https://localhost`

## Platform Management

### Primary Method: System Tray

The easiest way to manage Kamiwaza is through the system tray icon. Right-click the Kamiwaza icon in your system tray to access all management options.

### Alternative Method: Command Line

From PowerShell or Command Prompt:

```
# Start Kamiwaza
wsl -d kamiwaza -- kamiwaza start

# Stop Kamiwaza
wsl -d kamiwaza -- kamiwaza stop

# Restart Kamiwaza
wsl -d kamiwaza -- kamiwaza restart

# Check status
wsl -d kamiwaza -- kamiwaza status
```

## Troubleshooting

### Common Issues

#### Installation Fails with "WSL not found"

- Ensure WSL is installed: `wsl --install`
- Restart computer after WSL installation
- Verify with: `wsl --version`

#### Ubuntu-24.04 Distribution Not Found

- The installer will automatically download and install Ubuntu 24.04 if needed
- If installation fails, check existing distributions: `wsl --list --verbose`
- Re-run the installer if necessary

### Memory Allocation Errors

- Reduce memory allocation in installer
- Ensure sufficient free RAM on system
- Close other memory-intensive applications

### GPU Detection Issues

- Ensure latest GPU drivers are installed
- Check Windows version supports GPU passthrough
- Confirm GPU compatibility (NVIDIA or Intel Arc only)

### Network Access Problems

- Check Windows Firewall settings
- Verify ports 61100-61299 are available
- Try accessing `https://localhost` instead of `http://`

## Getting Help

### Check Installation Logs

**WSL logs:**

```
wsl -d kamiwaza -- journalctl -t kamiwaza-install
```

**Windows logs:**

- Check Event Viewer → Applications

Installation and other logs should also be located on your Windows device at: `C:\Users\[USER]\AppData\Local\Kamiwaza\logs`

### GPU Status Check

```
wsl -d kamiwaza -- /usr/local/bin/kamiwaza_gpu_status.sh
```

### Support Contact

- **Technical Support**: Contact our support team
- **License Issues**: Contact our support team

## Uninstallation

To completely remove Kamiwaza:

### Option 1: Use Windows Settings

1. Settings → Apps → Find "Kamiwaza Installer" → Uninstall

### Option 2: Use Start Menu shortcut

1. Start Menu → Kamiwaza → "Cleanup WSL (Uninstall)"

### Option 3: Manual cleanup if needed

```
wsl --unregister kamiwaza
```

---

**Last Updated**: September 3th, 2025

# Kamiwaza Offline Installation Guide for Red Hat Enterprise Linux

## Overview

Kamiwaza supports offline installation for air-gapped RHEL environments where internet access is restricted or unavailable. The offline installer includes:

- **Pre-packaged NVM + Node.js** (version 22.11.0)
- **Frontend node_modules** (if available during build)
- **Docker images** (containerized services)
- **Python wheels** (pip dependencies)
- **System dependencies** (RPM packages)

---

## Dependencies

The following dependencies are required:

- net-tools
- gcc-c++
- nodejs
- npm
- jq
- pkgconfig
- fontconfig-devel
- freetype-devel
- libX11-devel
- libXrender-devel
- libXext-devel
- libpng-devel
- libSM-devel

- pixman-devel
- libxcb-devel
- glib2-devel
- python3-devel
- libffi-devel
- gtk3-devel
- ca-certificates
- curl
- libcurl-devel
- cmake
- gnupg2
- iptables
- pciutils
- dos2unix
- unzip
- coreutils
- systemd
- wget
- make
- gcc
- openssl
- sqlite
- ncurses-libs
- readline
- libffi
- xz-libs
- expat
- tk
- zlib-devel
- bzip2-devel
- openssl-devel
- ncurses-devel
- sqlite-devel

- readline-devel
- tk-devel
- xz-devel
- expat-devel
- libuuid-devel
- yum-utils
- device-mapper-persistent-data
- lvm2
- git
- python3.12
- python3.12-pip
- python3.12-devel

For systems that can temporarily connect to the internet, these dependencies can be installed via the command:

```
sudo dnf install -y net-tools gcc-c++ nodejs npm jq pkgconfig fontconfig-devel
freetype-devel libX11-devel libXrender-devel libXext-devel libpng-devel libSM-
devel pixman-devel libxcb-devel glib2-devel python3-devel libffi-devel gtk3-devel
ca-certificates curl libcurl-devel cmake gnupg2 iptables pciutils dos2unix unzip
coreutils systemd wget make gcc openssl sqlite ncurses-libs readline libffi xz-
libs expat tk zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel
readline-devel tk-devel xz-devel expat-devel libuuid-devel yum-utils device-
mapper-persistent-data lvm2 git python3.12 python3.12-pip python3.12-devel
```

---

## For Users: Installing Offline

### Step 1: Transfer Files to Target System

You should receive these files from your builder:

```
kamiwaza-[version]-offline.rpm          # Main installer package
kamiwaza-requirements-export.zip        # Additional dependencies (optional)
```

**Transfer Methods:**

- USB drive/removable media
- Secure file transfer (scp, rsync)
- Physical media delivery

## Step 2: Basic Installation

```
# Install the RPM package
sudo rpm -i kamiwaza-[version]-offline.rpm

# The installer will automatically detect offline mode and use bundled resources
```

## Step 3: Configure Environment

After installation, configure the environment settings in `/etc/kamiwaza/env.sh`:

```
# Edit the environment configuration file
sudo nano /etc/kamiwaza/env.sh

# Add the following required settings:
export OFFLINE_MODE=true
export KAMIWAZA_LICENSE_KEY=""  # Add your license key here
```

**Important Notes:**

- `OFFLINE_MODE=true` enables offline installation mode, preventing internet downloads
- Replace `KAMIWAZA_LICENSE_KEY=""` with your actual license key provided by Kamiwaza
- This file is sourced automatically by Kamiwaza services on startup

## Step 4: Verification

```
kamiwaza status
```

or to watch status:

```
kamiwaza status -w
```

Once confirmed, Kamiwaza is installed!

## File Locations

| Component | Installed Location | Purpose |
|---|---|---|
| Main Application | `/opt/kamiwaza/` | Core application files |
| Configuration | `/etc/kamiwaza/` | Runtime configuration |
| Data Storage | `/var/lib/kamiwaza/` | Models, databases, logs |
| Offline Resources | `/usr/share/kamiwaza/` | Bundled dependencies |
| Service Scripts | `/usr/bin/kamiwaza*` | Command-line tools |
| Log Files | `/var/log/kamiwaza/` | Application logs |

---

## Network Ports

| Service | Port | Purpose |
|---|---|---|
| Web Interface | 3000 | Main UI |
| API Server | 8000 | REST API |
| Ray Dashboard | 8265 | Ray monitoring |
| Ray Client | 10001 | Ray cluster comm |
| Traefik | 80/443 | Reverse proxy |

# Windows GPU Setup Guide

## Overview

Kamiwaza supports hardware acceleration on Windows through WSL2 with the following GPU configurations:

- **NVIDIA GPUs** (RTX series, GTX series, Quadro series)
- **Intel Arc GPUs** (A3xx, A5xx, A7xx series)
- **Intel Integrated GPUs** (UHD Graphics, Iris Xe)

## Prerequisites

### System Requirements

- Windows 11 (Build 22000 or later)
- WSL2 enabled and updated
- Latest GPU drivers installed
- Compatible GPU hardware

### WSL2 Requirements

- WSL2 kernel version 5.10.60.1 or later
- Windows 11 with GPU virtualization support
- GPU drivers with WSL2 compatibility

## NVIDIA GPU Setup

### Supported Hardware

- **RTX 40 Series**: RTX 4090, RTX 4080, RTX 4070 Ti, RTX 4070, RTX 4060 Ti, RTX 4060
- **RTX 30 Series**: RTX 3090, RTX 3080, RTX 3070, RTX 3060 Ti, RTX 3060
- **RTX 20 Series**: RTX 2080 Ti, RTX 2080, RTX 2070, RTX 2060

- **GTX 16 Series**: GTX 1660 Ti, GTX 1660, GTX 1650
- **GTX 10 Series**: GTX 1080 Ti, GTX 1080, GTX 1070, GTX 1060

## Driver Requirements

- **Minimum**: NVIDIA Driver 470.82 or later
- **Recommended**: NVIDIA Driver 535.98 or later
- **Latest**: Download from NVIDIA Driver Downloads (https://www.nvidia.com/Download/index.aspx)

## Installation Steps

### 1. Install NVIDIA Drivers

1. Download the latest driver for your GPU
2. Run the installer as Administrator
3. Restart your computer
4. Verify installation: `nvidia-smi` in Command Prompt

### 2. Install NVIDIA CUDA Toolkit for WSL

```
# In WSL (Ubuntu 24.04)
wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/cuda-
keyring_1.1-1_all.deb
sudo dpkg -i cuda-keyring_1.1-1_all.deb
sudo apt-get update
sudo apt-get -y install cuda-toolkit-12-4
```

### 3. Verify GPU Access in WSL

```
# Check if GPU is visible
nvidia-smi

# Expected output:
# +-----------------------------------------------------------------------------+
# | NVIDIA-SMI 535.98                 Driver Version: 535.98                    |
# |-------------------------------+----------------------+----------------------+
# | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
# | Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
# |                               |                      |               MIG M. |
# |===============================+======================+======================|
# |   0  NVIDIA GeForce RTX 4090  On   | 00000000:01:00.0  Off |
N/A |
# |  0%   45C    P8    25W /  450W |      0MiB /  24576MiB |      0%      Default
|
# |                               |                      |                  N/A |
# +-------------------------------+----------------------+----------------------+
```

## Configuration Files

### .wslconfig (Windows)

```
[wsl2]
gpuSupport=true
memory=16GB
processors=8
```

### Environment Variables (WSL)

```
# Add to ~/.bashrc
export CUDA_HOME=/usr/local/cuda
export PATH=$PATH:$CUDA_HOME/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUDA_HOME/lib64
```

# Intel Arc GPU Setup

## Supported Hardware

- **Arc A7 Series**: A770, A750
- **Arc A5 Series**: A580, A570, A560, A550
- **Arc A3 Series**: A380, A370, A350, A310

## Driver Requirements

- **Minimum**: Intel Arc Driver 31.0.101.4502 or later
- **Recommended**: Latest Intel Arc Driver
- **Download**: Intel Arc Driver Downloads (https://www.intel.com/content/www/us/en/download/785597/intel-arc-iris-xe-graphics-whql-windows.html)

## Installation Steps

### 1. Install Intel Arc Drivers

1. Download the latest Intel Arc driver
2. Run the installer as Administrator
3. Restart your computer
4. Verify installation in Device Manager

### 2. Install Intel OpenCL Runtime and oneAPI (Recommended)

For optimal Intel GPU performance, install Intel's oneAPI toolkit:

```
# In WSL (Ubuntu 24.04)
# Add Intel's GPG key
wget -O- https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-
PRODUCTS.PUB | \
  gpg --dearmor | sudo tee /usr/share/keyrings/oneapi-keyring.gpg > /dev/null

# Add the oneAPI repository
echo "deb [signed-by=/usr/share/keyrings/oneapi-keyring.gpg]
https://apt.repos.intel.com/oneapi all main" | \
  sudo tee /etc/apt/sources.list.d/oneAPI.list

# Update and install Intel OpenCL runtime and oneAPI
sudo apt update
sudo apt install -y intel-opencl-icd intel-basekit

# Configure permissions
sudo usermod -a -G render $USER
newgrp render
```

### 3. Alternative: Install OpenCL Runtime Only

If you prefer not to install the full oneAPI toolkit:

```
# Install OpenCL loader and tools
sudo apt-get update
sudo apt-get install -y ocl-icd-libopencl1 ocl-icd-opencl-dev opencl-headers
clinfo

# Add Intel Graphics PPA for latest drivers
sudo apt-get install -y software-properties-common
sudo add-apt-repository -y ppa:kobuk-team/intel-graphics
sudo apt-get update
sudo apt-get install -y libze-intel-gpu1 libze1 intel-opencl-icd
```

### 4. Verify GPU Access in WSL

```
# Check OpenCL availability
clinfo | grep "Platform Name"

# Check GPU devices
clinfo | grep "Device Name"

# Expected output:
# Platform Name                                  Intel(R) OpenCL
# Device Name                                    Intel(R) Arc(TM) A770 Graphics
```

## Configuration Files

### .wslconfig (Windows)

```
[wsl2]
gpuSupport=true
memory=16GB
processors=8
```

### Environment Variables (WSL)

```
# Add to ~/.bashrc
export INTEL_OPENCL_CONFIG=/etc/OpenCL/vendors/intel.icd

# For oneAPI users, source the environment
echo 'source /opt/intel/oneapi/setvars.sh' >> ~/.bashrc
```

# Intel Integrated GPU Setup

## Supported Hardware

- **12th Gen Intel**: UHD Graphics 730, UHD Graphics 770
- **13th Gen Intel**: UHD Graphics 770, UHD Graphics 730
- **14th Gen Intel**: UHD Graphics 770, UHD Graphics 730
- **Intel Iris Xe**: Integrated graphics in 11th-14th gen processors

## Driver Requirements

- **Minimum**: Intel Graphics Driver 30.0.101.1190 or later
- **Recommended**: Latest Intel Graphics Driver
- **Download**: Intel Graphics Driver Downloads (https://www.intel.com/content/www/us/en/download/785597/intel-arc-iris-xe-graphics-whql-windows.html)

## Installation Steps

### 1. Install Intel Graphics Drivers

1. Download the latest Intel Graphics driver
2. Run the installer as Administrator
3. Restart your computer
4. Verify installation in Device Manager

### 2. Install Intel OpenCL Runtime

```
# In WSL (Ubuntu 24.04)
sudo apt-get update
sudo apt-get install -y intel-opencl-icd
```

### 3. Verify GPU Access in WSL

```
# Check OpenCL availability
clinfo | grep "Platform Name"

# Check GPU devices
clinfo | grep "Device Name"

# Expected output:
# Platform Name                                   Intel(R) OpenCL
# Device Name                                     Intel(R) UHD Graphics 770
```

# GPU Detection Scripts

## Automatic Detection (PowerShell)

```
# detect_gpu.ps1
$gpuInfo = Get-WmiObject -Class Win32_VideoController | Select-Object Name,
AdapterRAM, DriverVersion

foreach ($gpu in $gpuInfo) {
    if ($gpu.Name -match "NVIDIA") {
        Write-Host "NVIDIA GPU detected: $($gpu.Name)"
        # Run NVIDIA setup
    }
    elseif ($gpu.Name -match "Intel.*Arc") {
        Write-Host "Intel Arc GPU detected: $($gpu.Name)"
        # Run Intel Arc setup
    }
    elseif ($gpu.Name -match "Intel.*UHD|Intel.*Iris") {
        Write-Host "Intel Integrated GPU detected: $($gpu.Name)"
        # Run Intel Integrated setup
    }
}
```

## GPU Setup Scripts

### NVIDIA Setup (setup_nvidia_gpu.sh)

```
#!/bin/bash
# setup_nvidia_gpu.sh

echo "Setting up NVIDIA GPU acceleration..."

# Install CUDA toolkit
sudo apt-get update
sudo apt-get install -y cuda-toolkit-12-4

# Configure environment
echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
echo 'export PATH=$PATH:$CUDA_HOME/bin' >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUDA_HOME/lib64' >> ~/.bashrc

# Test GPU access
nvidia-smi

echo "NVIDIA GPU setup complete!"
```

### Intel Arc Setup (setup_intel_arc_gpu.sh)

```
#!/bin/bash
# setup_intel_arc_gpu.sh

echo "Setting up Intel Arc GPU acceleration..."

# Install oneAPI for optimal performance
wget -O- https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-
PRODUCTS.PUB | \
  gpg --dearmor | sudo tee /usr/share/keyrings/oneapi-keyring.gpg > /dev/null

echo "deb [signed-by=/usr/share/keyrings/oneapi-keyring.gpg]
https://apt.repos.intel.com/oneapi all main" | \
  sudo tee /etc/apt/sources.list.d/oneAPI.list

sudo apt update
sudo apt install -y intel-opencl-icd intel-basekit

# Configure permissions
sudo usermod -a -G render $USER
newgrp render

# Configure environment
echo 'source /opt/intel/oneapi/setvars.sh' >> ~/.bashrc
echo 'export INTEL_OPENCL_CONFIG=/etc/OpenCL/vendors/intel.icd' >> ~/.bashrc

# Test GPU access
clinfo | grep "Device Name"

echo "Intel Arc GPU setup complete!"
```

### Intel Integrated Setup (setup_intel_integrated_gpu.sh)

```
#!/bin/bash
# setup_intel_integrated_gpu.sh

echo "Setting up Intel Integrated GPU acceleration..."

# Install OpenCL runtime
sudo apt-get update
sudo apt-get install -y intel-opencl-icd

# Configure environment
echo 'export INTEL_OPENCL_CONFIG=/etc/OpenCL/vendors/intel.icd' >> ~/.bashrc

# Test GPU access
clinfo | grep "Device Name"

echo "Intel Integrated GPU setup complete!"
```

## Advanced Intel GPU Setup for AI Workloads

### Building llama.cpp with Intel GPU Support

For optimal Intel GPU performance with AI models, build llama.cpp with SYCL support:

```
# Install build dependencies
sudo apt-get install -y build-essential cmake libcurl4-openssl-dev

# Clone llama.cpp
git clone https://github.com/ggerganov/llama.cpp.git
cd llama.cpp

# Source oneAPI environment (required for SYCL build)
source /opt/intel/oneapi/setvars.sh

# Build with SYCL support
rm -rf build
mkdir -p build && cd build
cmake .. -DGGML_SYCL=ON -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx
make -j$(nproc)
```

### Testing Intel GPU Acceleration

```
# Download a sample model
mkdir -p ../models
cd ../models
wget https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct-GGUF/resolve/main/qwen2.5-0.5b-instruct-q8_0.gguf

# Test inference with GPU offloading
cd ../build
source /opt/intel/oneapi/setvars.sh
./bin/llama-cli \
    -m ../models/qwen2.5-0.5b-instruct-q8_0.gguf \
    -p "Hello, how are you?" \
    -n 128 \
    -ngl 999  # Offload all layers to GPU
```

## Troubleshooting

### Common GPU Issues

#### GPU Not Detected in WSL

```
# Check WSL version
wsl --list --verbose

# Ensure WSL2 is being used
wsl --set-version Ubuntu-24.04 2

# Check GPU support
wsl --status
```

#### Driver Compatibility Issues

1. **Update Windows**: Ensure Windows 11 is fully updated
2. **Update WSL**: `wsl --update`
3. **Reinstall drivers**: Remove and reinstall GPU drivers
4. **Check compatibility**: Verify GPU supports WSL2 virtualization

### Intel GPU Specific Issues

```
# Check OpenCL installation
clinfo

# Verify oneAPI environment (if installed)
source /opt/intel/oneapi/setvars.sh
sycl-ls

# Check permissions
groups $USER
# Should show 'render' in the list
```

### Performance Issues

1. **Memory allocation**: Increase WSL memory in .wslconfig
2. **Processor allocation**: Allocate more CPU cores
3. **GPU memory**: Ensure sufficient GPU VRAM
4. **Background processes**: Close unnecessary applications

## GPU Status Verification

### NVIDIA GPU

```
# Check GPU status
nvidia-smi

# Check CUDA availability
nvcc --version

# Test CUDA functionality
cuda-install-samples-12.4.sh ~
cd ~/NVIDIA_CUDA-12.4_Samples/1_Utilities/deviceQuery
make
./deviceQuery
```

### Intel GPU

```
# Check OpenCL availability
clinfo

# Check GPU information
lspci | grep -i vga

# Test OpenCL functionality
sudo apt-get install -y ocl-icd-opencl-dev
```

## Performance Optimization

### WSL Configuration (.wslconfig)

```
[wsl2]
gpuSupport=true
memory=32GB
processors=16
swap=8GB
localhostForwarding=true
```

### Environment Optimization

```
# Add to ~/.bashrc
export CUDA_CACHE_DISABLE=0
export CUDA_CACHE_MAXSIZE=1073741824
export INTEL_OPENCL_CONFIG=/etc/OpenCL/vendors/intel.icd

# For oneAPI users
echo 'source /opt/intel/oneapi/setvars.sh' >> ~/.bashrc
```

### GPU Memory Management

- **NVIDIA**: Use `nvidia-smi` to monitor GPU memory usage
- **Intel**: Monitor through Windows Task Manager
- **Optimization**: Close unnecessary GPU applications

## Support and Resources

### Official Documentation

- NVIDIA CUDA Documentation (https://docs.nvidia.com/cuda/)
- Intel OpenCL Documentation (https://www.intel.com/content/www/us/en/developer/tools/opencl/overview.html)
- Intel oneAPI Documentation (https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html)
- Microsoft WSL GPU Support (https://docs.microsoft.com/en-us/windows/wsl/tutorials/gpu-compute)

### Community Resources

- NVIDIA Developer Forums (https://forums.developer.nvidia.com/)
- Intel Community Forums (https://community.intel.com/)
- WSL GitHub Issues (https://github.com/microsoft/WSL/issues)

### Troubleshooting Tools

- **GPU-Z**: Detailed GPU information and monitoring
- **MSI Afterburner**: GPU monitoring and overclocking
- **HWiNFO**: Comprehensive system information
- **Windows Performance Monitor**: System performance analysis

---

**Last Updated**: October 3rd, 2025 **Version**: Compatible with Kamiwaza v0.5.1 **Support**: Contact our support team

# Quickstart

Get up and running with Kamiwaza in just a few minutes! This guide will walk you through starting the platform, deploying your first AI model, and launching a real application from the App Garden.

## Prerequisites

Before you begin, make sure you have:

- Kamiwaza installed and configured (see our Installation Guide)
- At least 16GB of available RAM
- A stable internet connection for downloading models

## Step 1: Start Kamiwaza

First, let's get the Kamiwaza platform running on your system.

### For Community Edition (Ubuntu .deb package)

If you installed via the .deb package, Kamiwaza should start automatically as a system service. You can check the status with:

```
kamiwaza status
```

If it's not running, start it with:

```
kamiwaza start
```

### For Manual Installations

Navigate to your Kamiwaza installation directory and start the platform:

```
cd /path/to/kamiwaza
bash startup/kamiwazad.sh start
```

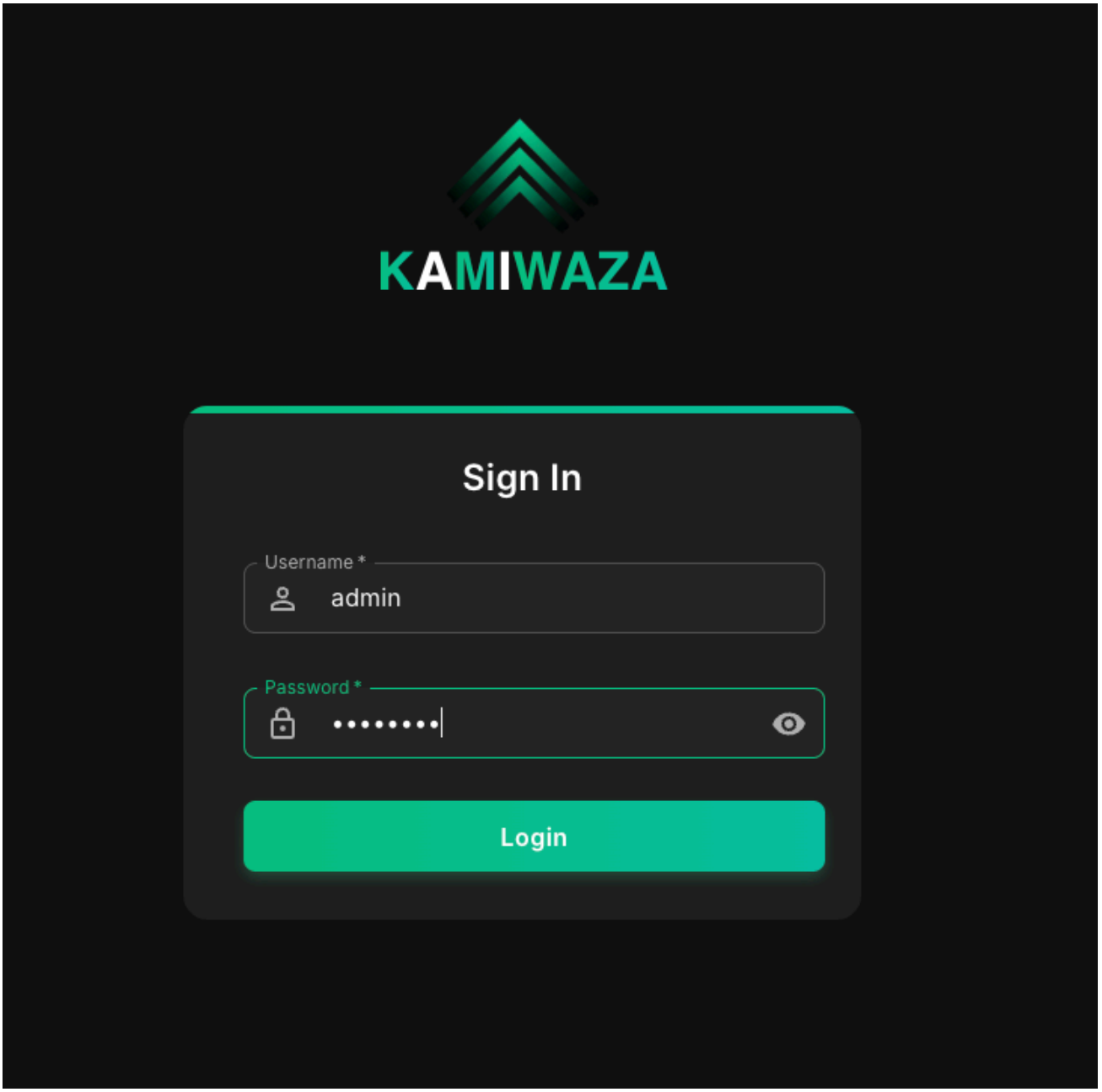
### Verify Kamiwaza is Running

Open your web browser and navigate to:

- **Frontend**: https://localhost (https://localhost)
- **API Documentation**: http://localhost/api/docs (http://localhost/api/docs)

You should see the Kamiwaza interface load successfully. Use the following credentials in the Sign In screen:

- **Username:** `admin`
- **Password:** `kamiwaza`



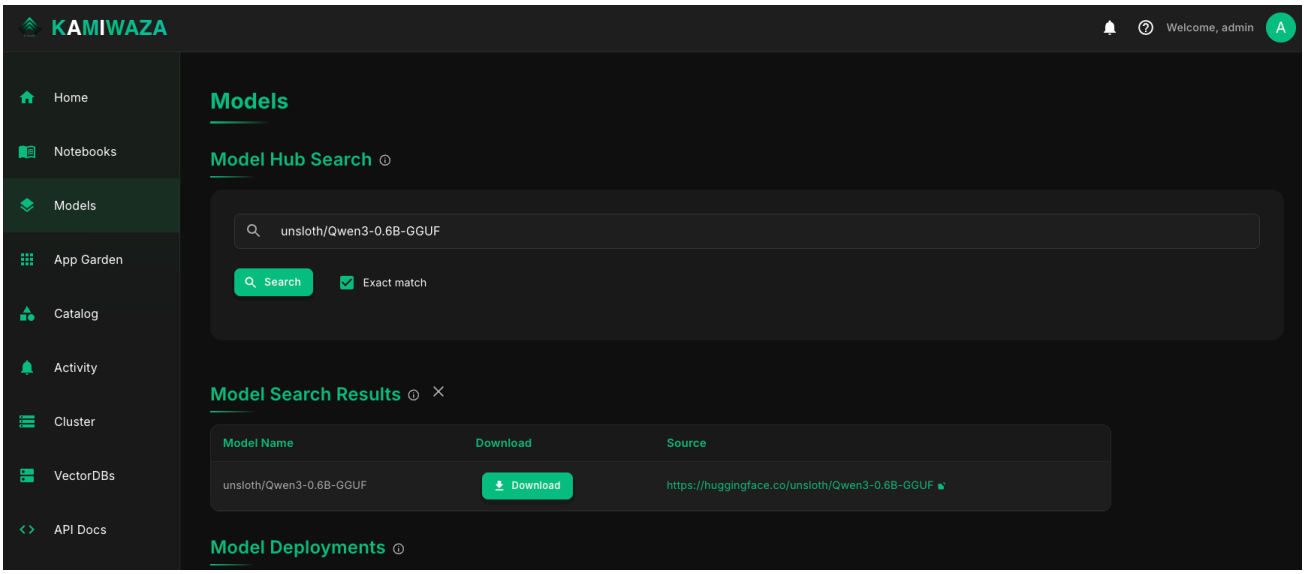

## Step 2: Deploy Your First Model

Now let's deploy a small, fast language model that's perfect for getting started.

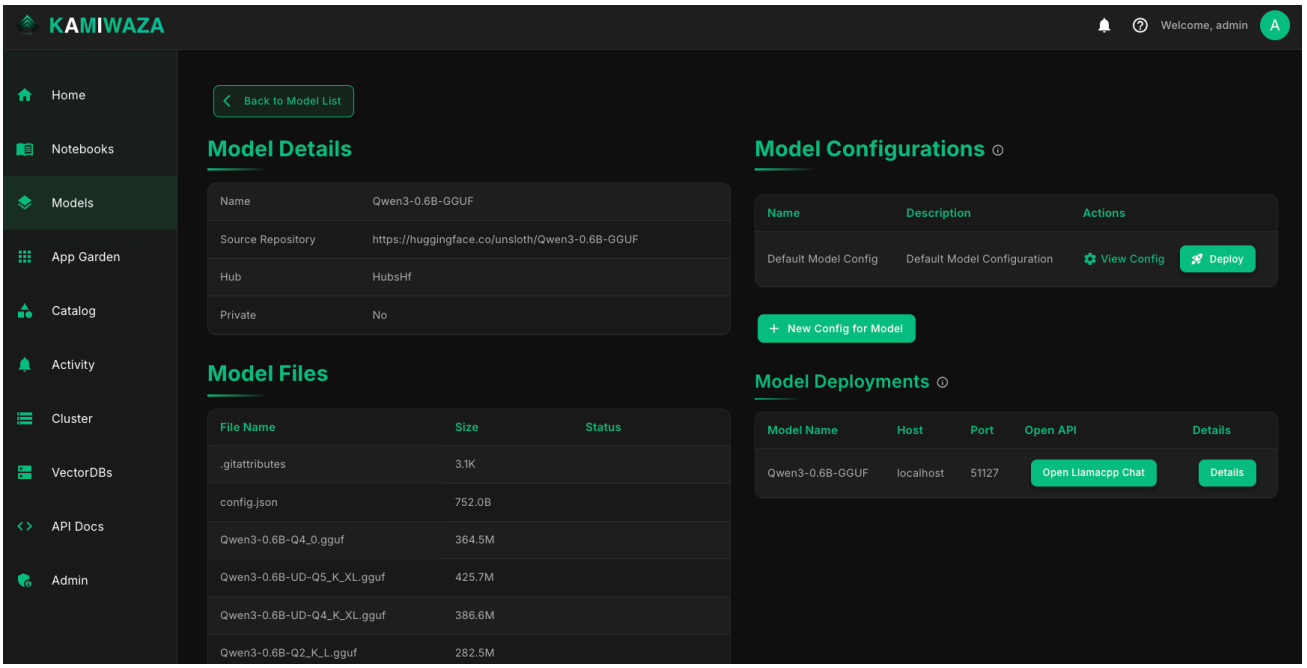
### Access the Models Section

1. In the Kamiwaza frontend, navigate to the **Models** section
2. Enter `unsloth/Qwen3-0.6B-GGUF` into the search field, click `Exact match`, then click Search.
   - Click the "Download" button, unselect all the files, just select the box next to `Qwen3-0.6B-Q6_K.gguf - 495.11 MB`

- Click "Download Selected Files".
- You should now see the file downloading in a new pop-up modal.




3. Deploy the model.
   - Scroll down to the "Your Models" section, and click the name of the model (`Qwen3-0.6B-GGUF`)
   - On the next screen, look for the "Deploy" button in the upper right (under Model Configurations) and click it.
   - After a few moments, you will see the model appear under "Model Deployments."




## Review

For the first deployment, we deployed a modern large language model:

- **Model**: `Qwen3 0.6B` (very light wieght, suitable for basic conversation)
  - Note that this is a GGUF-quantized version of the model suitable for CPU-based inference
- **Engine**: `llamacpp` (CPU-friendly)

## Step 3: Launch an AI Chatbot from the App Garden

Now let's use your deployed model in a real application from the App Garden.

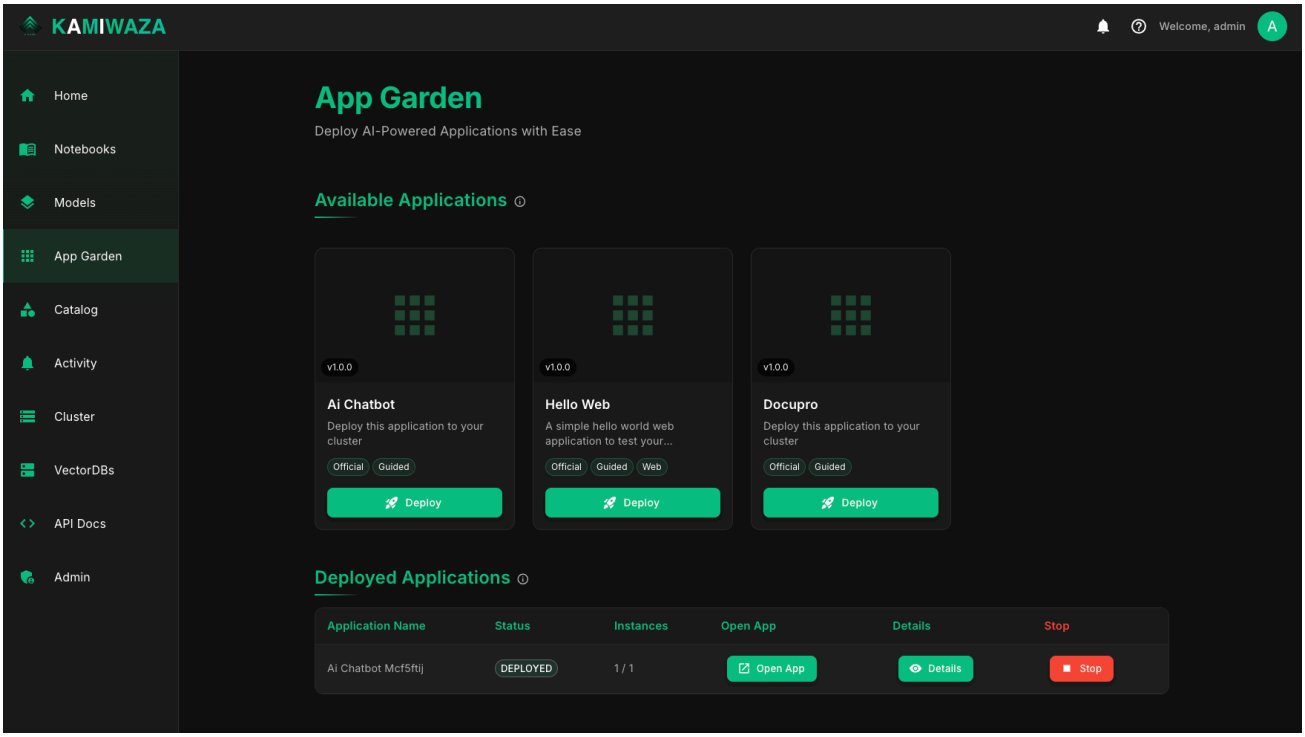
### Access the App Garden

1. Navigate to the **App Garden** section in the Kamiwaza interface
2. Browse the available applications
3. Look for the **"AI Chatbot"** app

**[Screenshot placeholder: App Garden interface showing available apps]**

### Configure and Launch the Chatbot

1. Click on the **AI Chatbot** app's "Deploy" button
2. In the configuration screen, keep the default configuration, and click **"Deploy"**
3. Click **"Launch App"**

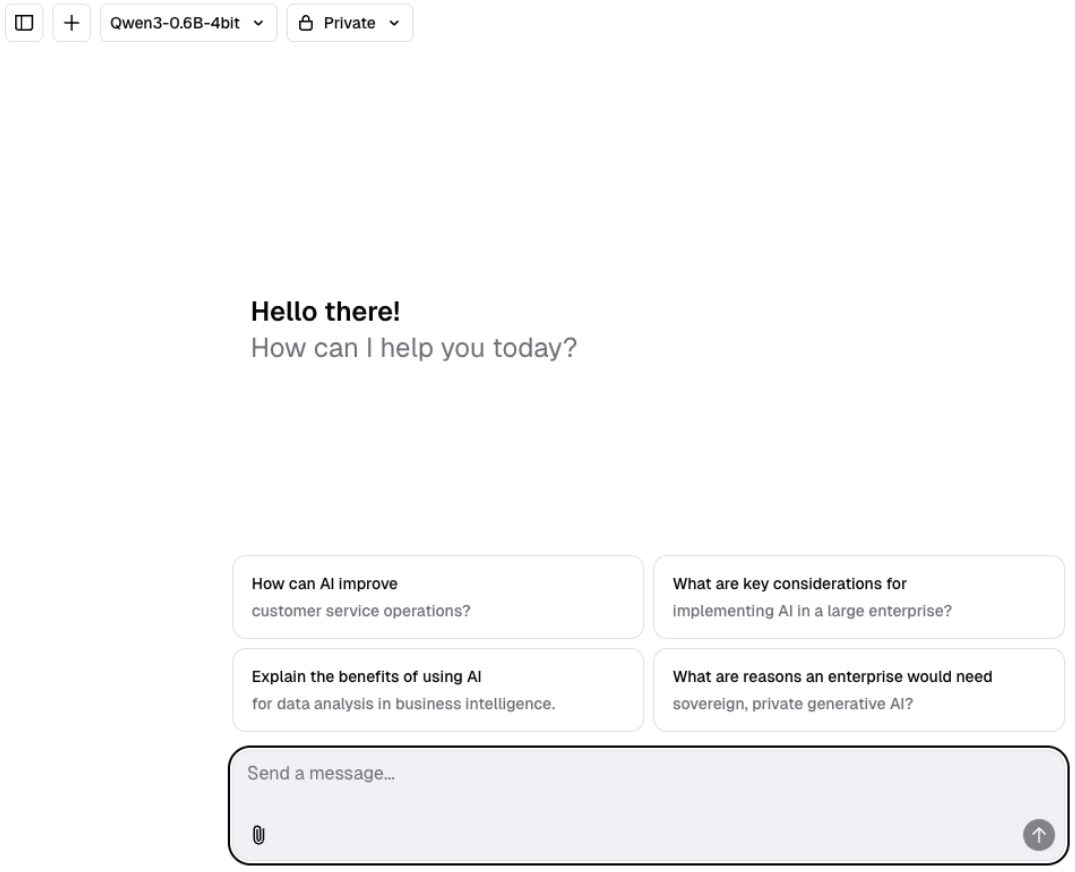


### Access Your Running Chatbot

- Once launched, click the **"Open App"** button in the deployed app listing,
- This will open the app in a new browser tab.
- It will automatically use your deployed model.

### Try Your Chatbot

You now have a fully functional AI-powered chat application!




## Next Steps

Congratulations! You've successfully: ✅ Started Kamiwaza
✅ Deployed your first AI model
✅ Launched a real application from the App Garden

### Explore More

Now that you have the basics down, here are some next steps to explore:

- **Try Different Models**: Deploy larger, more capable models
- **Explore More Apps**: Check out other applications in the App Garden
- **Learn the Architecture**: Understand how Kamiwaza works under the hood
- **Use the SDK**: Build custom applications using the Kamiwaza Python SDK
- **Set Up Your Data**: Connect your own data sources for RAG applications

### Need Help?

If you run into any issues:

- Check the troubleshooting section for common problems
- Join our Discord community (https://discord.gg/cVGBS5rD2U) for real-time help
- Contact our support team (https://portal.kamiwaza.ai/_hcms/mem/login?redirect_url=https%3A%2F%2Fportal.kamiwaza.ai%2Ftickets-view)

## What You've Learned

In this quickstart, you've experienced the core Kamiwaza workflow:

1. **Model Management**: How to deploy and test AI models
2. **App Garden**: How to launch pre-built applications
3. **Integration**: How models and apps work together seamlessly

This same pattern scales from simple chatbots to complex enterprise AI applications. Welcome to Kamiwaza!

# 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.

# 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’s 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

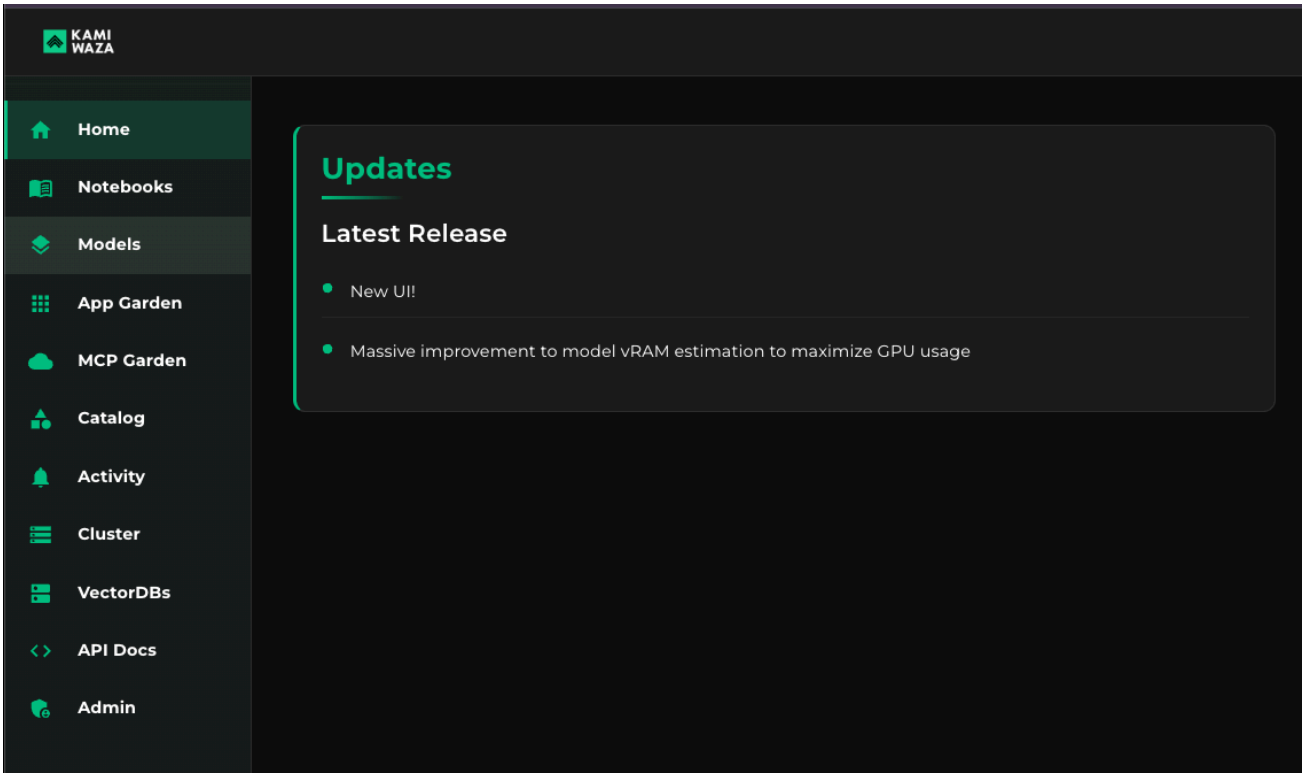
Need more control? Switch to Advanced Mode from settings to customize deployment parameters (quantization, memory, batching, prompt formatting, and more). Helpful guides:

- Deploying Models
- Downloading Models
- GUI Walkthrough

# GUI Walkthrough

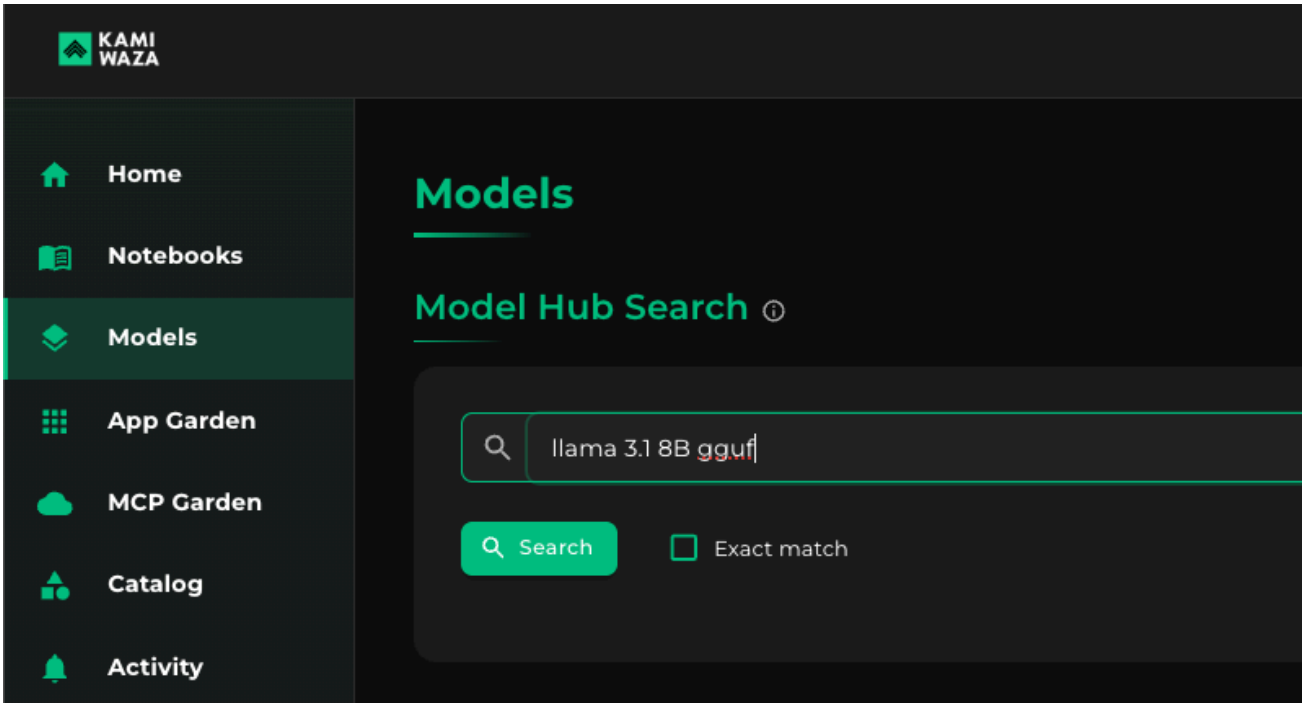
The following walkthrough is based on the Kamiwaza 1.0.0 user interface. We'll walk through navigating the Kamiwaza admin console to **search** for a model, **download** model files, and then **deploy and use** an inference endpoint.

## Step 1: Find and click the Models menu in the sidebar

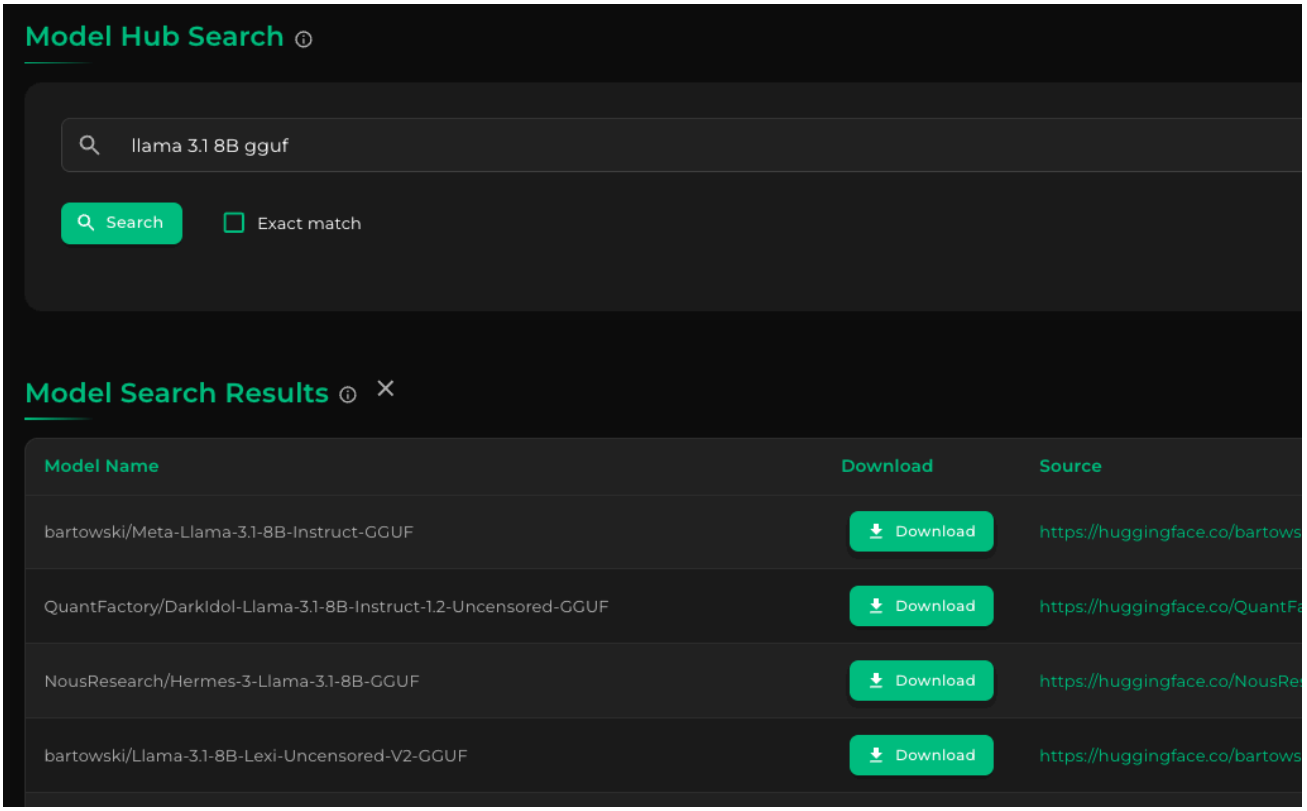



## Step 2: Under Model Hub Search, type keywords for your desired model

In this example, I'm looking for a GGUF of Llama3.1 8B Instruct, so I type a few of its keywords to narrow results.

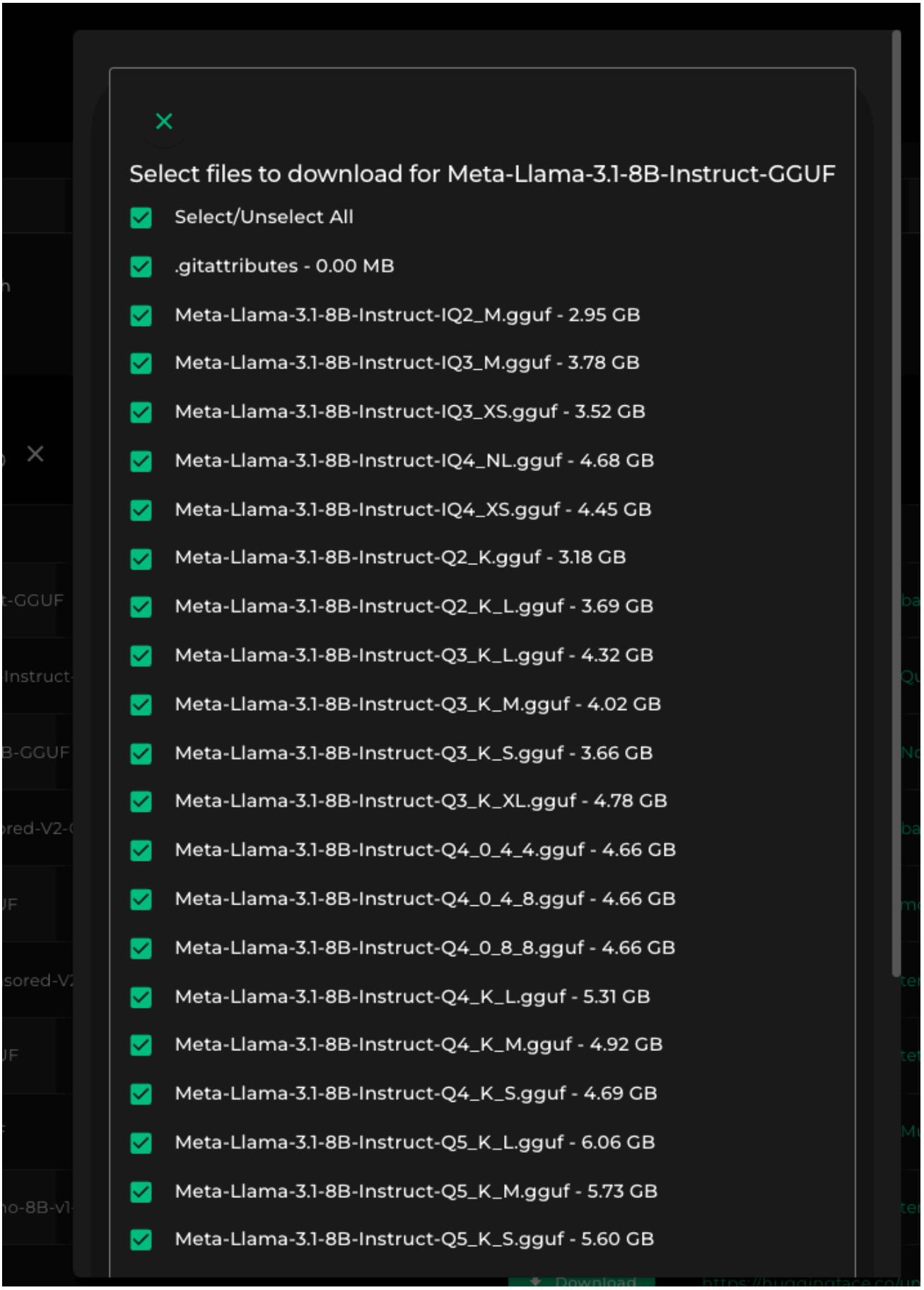



Clicking the **Search** button will show the results, like this:




## Step 3: Download models files from chosen model

From the results, let's choose the Bartowski GGUF. Clicking the **Download** button for *bartowski/Meta-Llama-3.1-8B-Instruct-GGUF* results in the following:



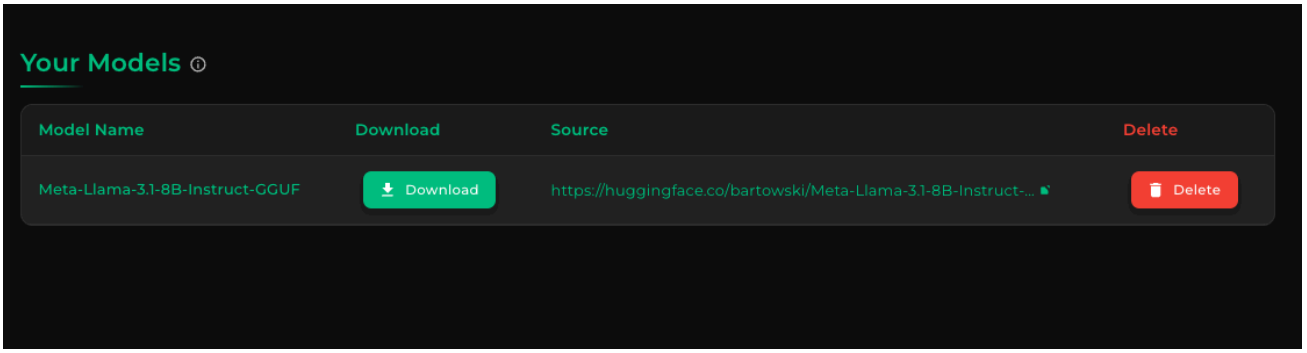
In this example, we are downloading GGUF models for a llamacpp inference engine deployment. Unlike normal Hugging Face models (safetensors), we only need to download one or a couple files - just the specific quantized model that we need.

For this example, we will 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.

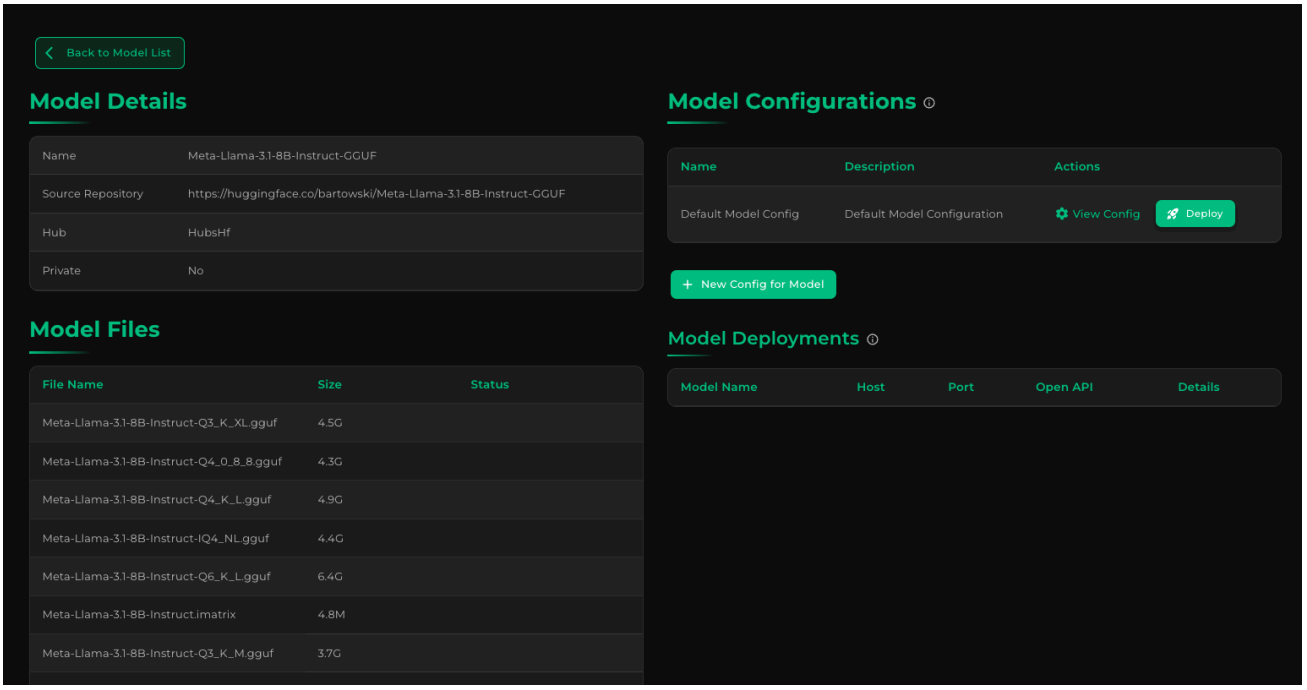
> NOTE: In normal Hugging Face models (non-GGUF), we usually need to download all files in order to serve a model (safetensors, config, tokenizer, etc), hence having everything pre-checked. We'd only need to scroll down and click the **Download Selected Files** button.

## 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'll see a screen that looks like this:

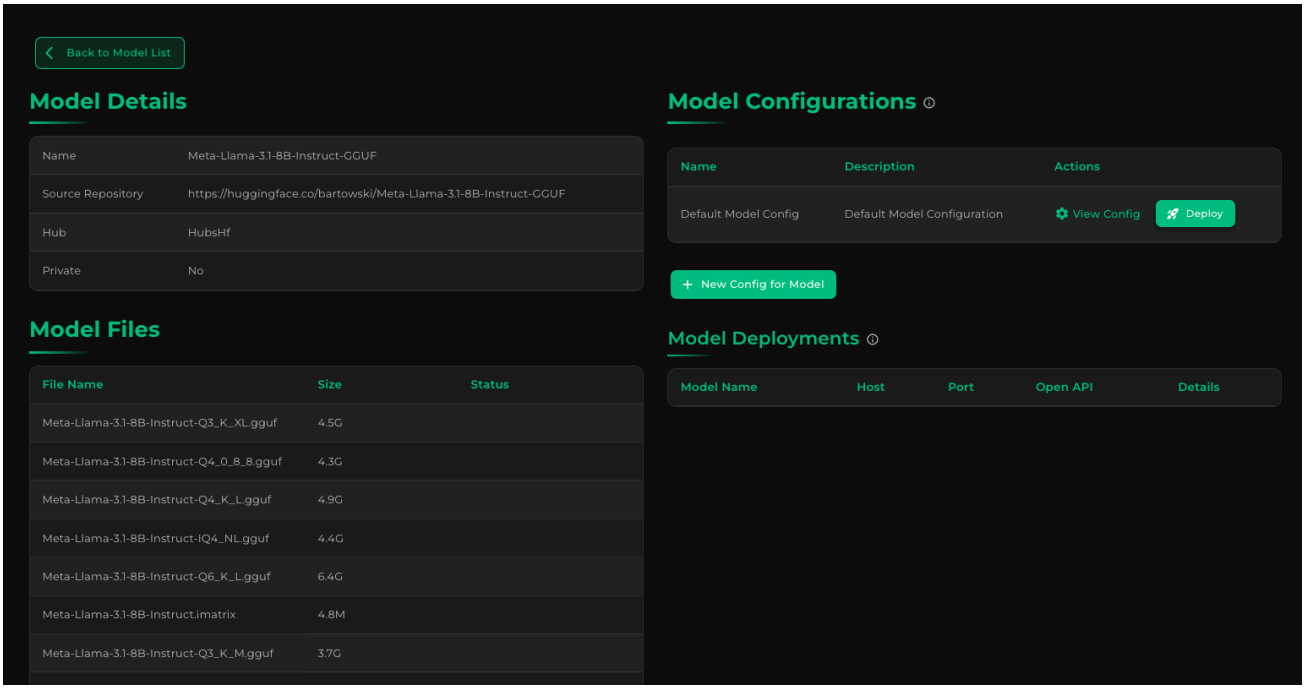



You'll notice most of the files under **Model Files** have an empty status, since we didn't download them. Scroll down to see if our 8-bit quantized GGUF is present, though:


And yep, as expected, our Q8_0-quantized GGUF is there. Since it is already marked as "downloaded", that means we can deploy it already.

## Step 5: Deploy a model for serving

We've already seen the **Deploy** button earlier, at the upper right area of the Model Details screen, under Model Configurations:

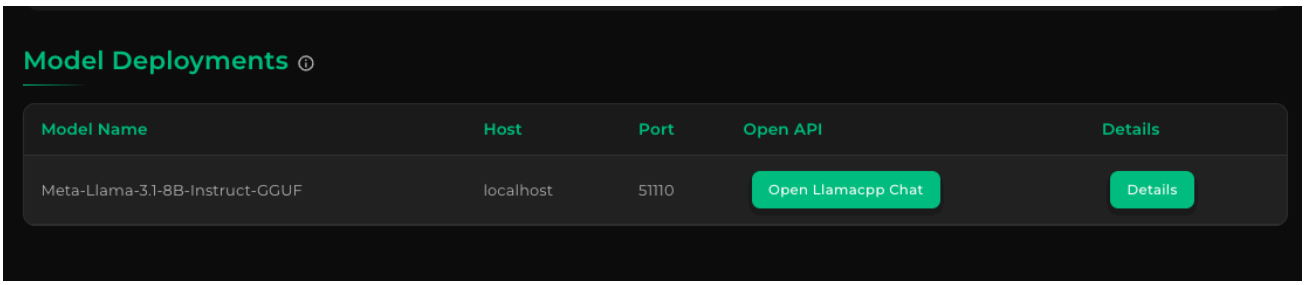



Kamiwaza creates a default config for us when we download a model, to simplify the deployment experience.

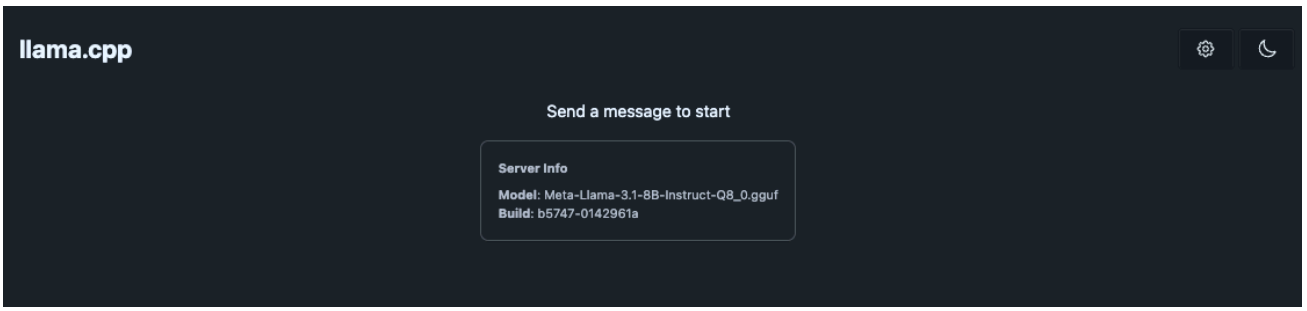
Click **"Deploy"** to launch an instance. When the spinner finishes, click **Back to Model List** to go back to the main Models screen.

## Step 6: Testing the deployed model endpoint

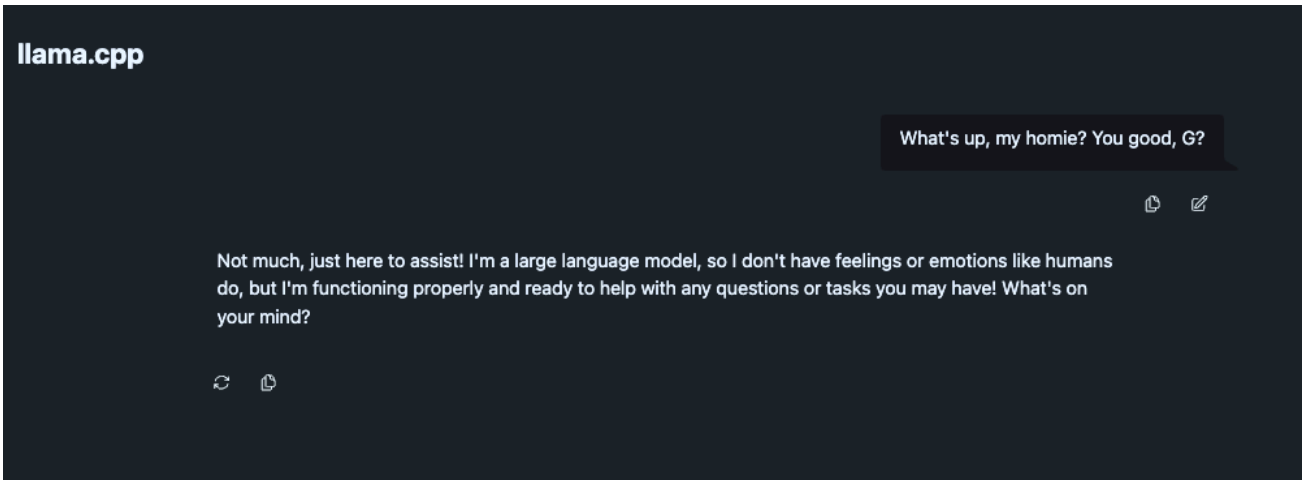
In the main Models screen, under Model Deployments, you will see a new entry for our recently deployed Llama 3.1 8B model:



To test our model, click "Open Llamacpp Chat". Your browser will open a chat application running against the deployed model:




Let's send a test message to validate that our LLM endpoint is working:




Perfect! If you see a response like that, then our inference endpoint is working as expected.

🎉 **You've 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.

# 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 a network port and configures the load balancer (Traefik) to route requests to the new model endpoint.
4. **Launch**: The selected engine is started. For `vLLM` on Linux, this is a Docker container. For `MLX` on macOS, it's a native process.
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

Kamiwaza wires the public port to Ray Serve for model traffic. Routes can be created immediately after launch; Ray Serve handles readiness internally. This is why you may see INITIALIZING briefly before DEPLOYED.

# 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? 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

# App Garden

App Garden lets you browse, deploy, and manage containerized applications from a curated catalog—all from the Kamiwaza UI. Apps are packaged with Docker Compose and deploy in a few clicks with sensible defaults.

## What is App Garden?

App Garden is a catalog of ready-to-run apps (dashboards, demo UIs, tools) that you can deploy to your Kamiwaza environment. It handles the container runtime, networking, and routing for you, so you focus on using the app, not wiring it up.

## Key Features

- **One-click deploy**: Launch apps directly from the catalog
- **Automatic routing**: Each app gets a stable URL via the built-in load balancer
- **Cross-platform**: Works on macOS, Windows, and Linux
- **AI-ready**: Apps can automatically connect to your deployed models (OpenAI-compatible)
- **Simple lifecycle**: Start, stop, and remove from the UI

## 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 default 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've 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.

## 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 won't start**: Retry Deploy. If it persists, Stop/Remove and deploy again.
- **Can't 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.

## Advanced Options

App customization is coming soon!

# Distributed Data Engine

*Sharpening our katana... stay tuned!* ⚔️

# 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'll 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

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'll 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

We're working on additional use case guides including:

### 🤖 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're looking for a specific use case that isn't covered yet:

- Join our Discord community (https://discord.gg/cVGBS5rD2U) to ask questions
- Explore the SDK documentation for programmatic approaches
- Check out the Models and App Garden sections for additional patterns

---

*Have a use case you'd like to see documented? Let us know on our Discord community (https://discord.gg/cVGBS5rD2U) or contact us (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'll Build

By the end of this guide, you'll 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)
- 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

First, we'll 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:

```
from kamiwaza_client import KamiwazaClient

client = KamiwazaClient(base_url="http://localhost:7777/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:

```
from kamiwaza_client import KamiwazaClient

client = KamiwazaClient(base_url="http://localhost:7777/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

```
# 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

Now we'll 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_client import KamiwazaClient

class RAGPipeline:
    def __init__(self, base_url="http://localhost:7777/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

Now we'll create the query interface that retrieves relevant documents and generates responses.

```
from typing import List, Dict
from kamiwaza_client import KamiwazaClient

class RAGQuery:
    def __init__(self, base_url="http://localhost:7777/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

Let's 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="http://localhost:7777/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="http://localhost:7777/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

- **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

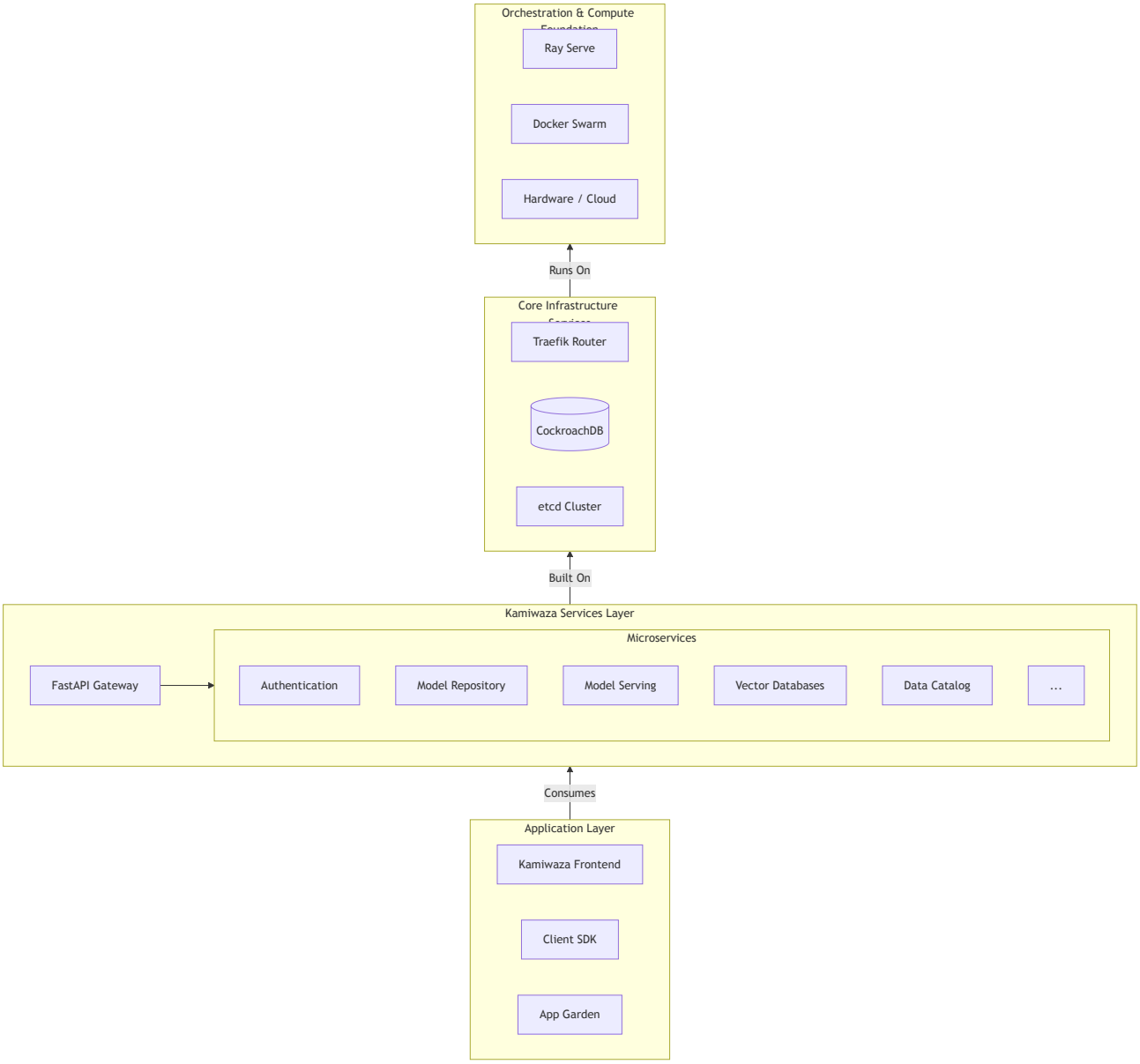
Your RAG pipeline is now ready to answer questions using your own documents! The combination of Kamiwaza's SDK with proper document processing creates a robust foundation for production RAG applications.

# Platform Architecture Overview

Kamiwaza is a modular, multi-layered platform designed for scalability, flexibility, and enterprise-grade performance. This document provides a high-level overview of its key architectural components and the technologies that power them.

## System Architecture Diagram

The diagram below illustrates the layered architecture of the Kamiwaza platform, from the underlying infrastructure to the user-facing applications. Each layer provides a distinct set of capabilities, creating a robust and maintainable system.




## Architecture Layers

### Application Layer

This is where users and developers interact with Kamiwaza.

- **Kamiwaza Frontend**: The primary web-based user interface for managing the platform.
- **Client SDK**: A Python SDK for programmatically interacting with Kamiwaza's APIs.
- **App Garden**: A platform for discovering and deploying pre-packaged AI applications and services.

### 🎯 Kamiwaza Services Layer

The core business logic of the platform, exposed via a central API gateway.

- **FastAPI Gateway**: A high-performance API gateway that coordinates all requests.
- **Microservices**: A suite of specialized services handling concerns like Authentication, Model Management, Vector DB abstraction (Milvus, Qdrant), and more.

### 🛠️ Core Infrastructure Services

The essential backend services that support the entire platform.

- **Traefik Router**: A powerful reverse proxy and load balancer that manages all inbound network traffic and provides SSL termination.
- **CockroachDB**: The primary database—a distributed, resilient SQL database for all application data.
- **etcd**: A distributed key-value store used for service discovery and critical configuration management across the cluster.

### 🔧 Orchestration & Compute Foundation

The base layer that provides the compute resources and orchestration to run everything else.

- **Ray Serve**: A scalable model-serving framework used for deploying and managing AI models.
- **Docker Swarm**: The container orchestration engine that manages the lifecycle of all containerized services.
- **Hardware / Cloud**: The physical servers or cloud instances (e.g., AWS, GCP, Azure) that provide the underlying CPU and GPU resources.

## Technology Stack

| Category | Technologies |
| --- | --- |
| **Backend** | Python 3.10, FastAPI, Ray, SQLAlchemy, Pydantic |
| **Frontend** | React 18, Material-UI, Tailwind CSS, Axios |
| **Databases** | CockroachDB, Milvus, Qdrant, etcd |
| **Infrastructure** | Docker Swarm, Traefik, DataHub, JupyterHub |

# Core Components & Concepts

The Kamiwaza platform is composed of several key components and conceptual systems that work together to provide a comprehensive AI orchestration solution. This page describes the most important of these building blocks.

## 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 integrates with industry-standard vector databases like **Milvus** and **Qdrant**.
- **Why it matters:** Vector databases are the engine behind powerful similarity searches, which are essential for Retrieval-Augmented Generation (RAG), recommendation engines, and other advanced AI applications. Kamiwaza's abstraction layer lets you choose the right database for your needs without changing your application code.

### Embeddings Management

- **What it is:** The process of generating, storing, and managing the vector embeddings for your data. Kamiwaza provides built-in services to automate the creation of embeddings using various open-source or custom models.
- **Why it matters:** Consistent and efficient embedding management is fundamental to the performance of any vector search-based application. By handling this automatically, Kamiwaza reduces a major source of complexity in building RAG pipelines.

### Data Catalog

- **What it is:** A centralized inventory of all your data assets. Kamiwaza integrates with **Acryl DataHub** to provide a single place to discover, understand, and govern your data.
- **Why it matters:** As AI systems grow, so does the data they consume. A data catalog provides crucial lineage tracking ("where did this data come from?") and discoverability, which is vital for enterprise governance, security, and scalability.

## Orchestration & Serving

### Orchestration Engine

- **What it is:** The "brain" of the platform that manages the flow of requests and coordinates tasks between different services. Kamiwaza uses frameworks like **Ray Serve** to handle this complex, distributed workload.
- **Why it matters:** The orchestrator ensures that AI requests are processed efficiently, scaled according to demand, and routed to the correct models and services. This is the key to building resilient, production-grade AI applications.

### Model Serving

- **What it is:** The process of taking a trained AI model and making it available for real-time inference via an API. Kamiwaza supports multiple high-performance serving engines like **vLLM**, **llama.cpp**, and **MLX**.
- **Why it matters:** Different models have different hardware needs (CPU vs. GPU). Kamiwaza's multi-engine support ensures you can run a diverse range of models and optimize for both performance and cost.

### API Gateway

- **What it is:** A single, unified entry point for all API requests to the platform. Kamiwaza uses **FastAPI** to create this gateway, which then routes requests to the appropriate internal microservice.
- **Why it matters:** A gateway simplifies development by providing a consistent interface for all platform services. It's also the ideal place to enforce cross-cutting concerns like 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're 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 (e.g., **Redis**, **Valkey**) that keeps frequently accessed data readily available, reducing the need to re-compute or re-fetch it from slower databases.
- **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 microservices, each handling a specific aspect of the AI platform's functionality. These services work together to provide a comprehensive AI orchestration platform that manages the entire lifecycle of AI models and applications.

## 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 ensures:

- **Separation of concerns** - Each service has a clear, focused responsibility
- **Scalability** - Services can be scaled independently based on demand
- **Maintainability** - Changes to one service don't affect others
- **Testability** - Each service can be tested in isolation

## Core Services Overview

### 🤖 Models Service

Manages the complete lifecycle of AI models including deployment, versioning, and serving. This service handles everything from model downloads to runtime management, supporting multiple serving engines like llama.cpp, vLLM, and Transformers.

### 🔍 Vector Database Service

Provides an abstraction layer over vector databases like Milvus and Qdrant, enabling efficient storage and retrieval of high-dimensional embeddings. Supports hybrid search, metadata filtering, and performance optimization.

### 📄 Retrieval Service

Powers RAG (Retrieval-Augmented Generation) pipelines and document search capabilities. Combines vector similarity with keyword search, provides reranking, and supports advanced query processing for contextual AI applications.

### 🧠 Embedding Service

Handles text embedding generation and storage, converting text into numerical representations for vector similarity searches. Supports multiple embedding models, batch processing, and intelligent caching strategies.

### 🔐 Authentication Service

Manages JWT-based authentication and integrates with various identity providers to secure platform access. Supports OAuth, SAML, multi-factor authentication, and role-based access control.

### 📊 Catalog Service

Integrates with Acryl DataHub to provide data cataloging and metadata management capabilities. Enables data discovery, lineage tracking, and governance across the AI platform.

### 📈 Activity Service

Provides comprehensive audit logging and metrics collection for monitoring platform usage and performance. Tracks user actions, system events, and provides real-time dashboards and alerting.

### 💬 Prompts Service

Manages a centralized library of prompt templates for consistent AI interactions across applications. Supports versioning, A/B testing, and performance tracking for prompt optimization workflows.

## Service Communication

All services communicate through:

- **FastAPI routers** for HTTP API endpoints
- **Ray Serve** for distributed computing and scaling
- **Shared databases** (CockroachDB, etcd) for state management
- **Message queues** for asynchronous processing

## Integration Patterns

Services are designed to work together seamlessly:

- **Models + Embedding** - Deploy embedding models for text vectorization
- **Embedding + VectorDB** - Store and retrieve high-dimensional embeddings
- **VectorDB + Retrieval** - Power semantic search and RAG pipelines
- **Retrieval + Prompts** - Combine context retrieval with optimized prompts
- **Activity + All Services** - Monitor and log all platform interactions

## Next Steps

To learn more about working with these services:

- Explore the Models and Distributed Data Engine documentation
- Build a complete RAG pipeline using multiple services
- Review the Platform Overview for architectural context
- Check out Use Cases for practical implementation examples

# Administrator Guide

## 1. Authentication & Access Control

Kamiwaza provides enterprise-grade authentication built on **Keycloak** with OpenID Connect (OIDC) and JWT token validation.

### 1.1 Authentication Architecture

```
User → Keycloak (IdP) → JWT Token → Traefik → ForwardAuth → API Services
                                          ↓
                                    [Validated] → Access Granted
                                          ↓
                                    [Rejected] → 401/403 Error
```

**Components:**

- **Keycloak**: Identity provider managing users, authentication, and token issuance
- **ForwardAuth Service**: Validates JWT tokens and enforces access policies
- **Traefik**: Reverse proxy routing requests through ForwardAuth middleware
- **RBAC Policy Engine**: YAML-based endpoint access control

### 1.2 Authentication Modes

Kamiwaza supports two operational modes:

| Mode | Use Case | Configuration |
|---|---|---|
| **With Authentication** | Production, staging, secure environments | `KAMIWAZA_USE_AUTH=true` |
| **Bypass Mode** | Local development, debugging | `KAMIWAZA_USE_AUTH=false` |

**To enable authentication:**

```
# In env.sh or environment
export KAMIWAZA_USE_AUTH=true
bash startup/kamiwazad.sh restart
```

⚠️ **Warning:** Bypass mode (`KAMIWAZA_USE_AUTH=false`) disables all authentication. Use only in secure development environments.

### 1.3 Token-Based Authentication

Kamiwaza uses **RS256 JWT tokens** with asymmetric cryptographic signatures.

**Token Lifecycle:**

1. **Acquisition**: User authenticates with Keycloak via username/password or SSO
2. **Validation**: ForwardAuth validates token signature against JWKS endpoint
3. **Authorization**: User roles checked against RBAC policy
4. **Expiration**: Access tokens expire (default: 1 hour), require refresh
5. **Revocation**: Logout invalidates tokens

**Token Delivery Methods:**

- HTTP `Authorization: Bearer <token>` header (recommended for APIs)
- Secure HTTP-only cookie (automatic for browser sessions)

---

## 2. User Management

### 2.1 Accessing Keycloak Admin Console

**Default Credentials** (change immediately in production):

- **URL:** http://localhost:8080 (http://localhost:8080) (or your configured Keycloak URL)
- **Username:** `admin`
- **Password:** Set via `KEYCLOAK_ADMIN_PASSWORD` environment variable

**Production Setup:**

```
# Set secure admin password in env.sh
export KEYCLOAK_ADMIN_PASSWORD="<strong-random-password>"
```

## 2.2 Creating User Accounts

**Via Keycloak Admin Console:**

1. Navigate to **Users** in left sidebar
2. Click **Add User**
3. Fill in required fields:
   - **Username** (required)
   - **Email** (required for password reset)
   - **First Name / Last Name** (optional)
4. Toggle **Email Verified** to `ON`
5. Click **Save**
6. Go to **Credentials** tab
7. Set temporary or permanent password
8. Assign roles (see Role Management below)

**Pre-configured Test Users:**

| Username | Password | Roles | Use Case |
|---|---|---|---|
| `testuser` | `testpass` | viewer | Read-only testing |
| `testadmin` | `testpass` | admin | Administrative testing |

⚠️ **Important:** Remove or secure test users before production deployment.

## 2.3 User Roles and Permissions

Kamiwaza defines three primary roles:

| Role | Permissions | Typical Users |
|---|---|---|
| **admin** | Full access: read, write, delete, configure | System administrators, platform operators |
| **user** | Standard access: read, write (no delete/admin) | Data scientists, developers, analysts |
| **viewer** | Read-only access | Auditors, observers, stakeholders |

**Assigning Roles:**

1. Navigate to **Users** → Select user
2. Go to **Role Mappings** tab
3. Under **Realm Roles**, select appropriate roles
4. Click **Add selected**
5. Changes take effect immediately (no logout required)

## 2.4 Password Policies

**Configuring Password Requirements:**

1. Navigate to **Realm Settings** → **Security Defenses** → **Password Policy**
2. Add policies:
   - **Minimum Length**: 12 characters (recommended)
   - **Uppercase Characters**: Require at least 1
   - **Lowercase Characters**: Require at least 1
   - **Digits**: Require at least 1
   - **Special Characters**: Require at least 1
   - **Not Username**: Prevent username as password
   - **Password History**: Prevent last 3 passwords
   - **Expire Password**: 90 days (recommended)

**Password Reset Flow:**

1. User clicks "Forgot Password" on login page
2. Keycloak sends password reset email
3. User follows link and sets new password
4. New password must meet policy requirements

⚠️ **Important:** Configure SMTP settings in Keycloak for email-based password reset to function.

---

## 3. Role-Based Access Control (RBAC)

### 3.1 RBAC Policy File

Access control is defined in **YAML policy files** that map endpoints to required roles.

**Default Location:**

- Host installs: `$KAMIWAZA_ROOT/config/auth_gateway_policy.yaml`
- Docker installs: Mounted at `/app/config/auth_gateway_policy.yaml`

**Policy File Structure:**

```yaml
version: 1
env: production
default_deny: true  # Block all endpoints unless explicitly allowed

roles:
  - id: admin
    description: "Full system access"
  - id: user
    description: "Standard user access"
  - id: viewer
    description: "Read-only access"

endpoints:
  # Model Management
  - path: "/api/models*"
    methods: ["GET"]
    roles: ["viewer", "user", "admin"]

  - path: "/api/models*"
    methods: ["POST", "PUT", "DELETE"]
    roles: ["user", "admin"]

  # Cluster Management (Admin-only)
  - path: "/api/cluster*"
    methods: ["*"]
    roles: ["admin"]

  # Vector Database (User and Admin)
  - path: "/api/vectordb*"
    methods: ["GET"]
    roles: ["viewer", "user", "admin"]

  - path: "/api/vectordb*"
    methods: ["POST", "PUT", "DELETE"]
    roles: ["user", "admin"]

  # Public endpoints (no auth required)
  - path: "/health"
    methods: ["GET"]
    roles: ["*"]  # Public

  - path: "/docs"
    methods: ["GET"]
    roles: ["*"]  # Public API documentation
```

## 3.2 Path Matching Rules

**Wildcard Patterns:**

- `*` matches zero or more characters within a path segment
- `**` matches across multiple path segments

- Patterns are case-sensitive

**Examples:**

- `/api/models*` matches `/api/models`, `/api/models/123`, `/api/models/search`
- `/api/*/health` matches `/api/models/health`, `/api/cluster/health`
- `/api/**` matches all paths under `/api/`

### 3.3 Hot Reload (No Restart Required)

The RBAC policy file is automatically reloaded when modified:

1. Edit `auth_gateway_policy.yaml`
2. Save the file
3. Changes take effect within seconds
4. Monitor logs for reload confirmation:

```
INFO: Policy reloaded successfully from /app/config/auth_gateway_policy.yaml
```

⚠️ **Important:** Invalid YAML syntax will prevent reload and retain the previous valid configuration.

### 3.4 Adding Custom Endpoints

**Example: Protecting a new analytics endpoint**

```yaml
endpoints:
  # Add new analytics endpoint
  - path: "/api/analytics/reports*"
    methods: ["GET"]
    roles: ["user", "admin"]

  - path: "/api/analytics/reports*"
    methods: ["POST", "DELETE"]
    roles: ["admin"]
```

**Testing Access Control:**

```
# Get token for viewer role (should be denied POST)
VIEWER_TOKEN=$(curl -s -X POST
http://localhost:8080/realms/kamiwaza/protocol/openid-connect/token \
  -d "grant_type=password" \
  -d "client_id=kamiwaza-platform" \
  -d "username=testuser" \
  -d "password=testpass" | jq -r .access_token)

# Test (expect 403 Forbidden)
curl -H "Authorization: Bearer $VIEWER_TOKEN" \
  -X POST http://localhost:7777/api/analytics/reports

# Get token for admin role (should succeed)
ADMIN_TOKEN=$(curl -s -X POST
http://localhost:8080/realms/kamiwaza/protocol/openid-connect/token \
  -d "grant_type=password" \
  -d "client_id=kamiwaza-platform" \
  -d "username=testadmin" \
  -d "password=testpass" | jq -r .access_token)

# Test (expect 200 OK)
curl -H "Authorization: Bearer $ADMIN_TOKEN" \
  -X POST http://localhost:7777/api/analytics/reports
```

---

## 4. Identity Provider Integration

### 4.1 Keycloak Configuration

**Realm:** `kamiwaza` **Client ID:** `kamiwaza-platform`

**Client Configuration Settings:**

| Setting | Value | Purpose |
|---|---|---|
| **Access Type** | Public (SPA) or Confidential (backend) | Authentication flow type |
| **Valid Redirect URIs** | `https://your-domain.com/*` | Allowed OAuth callback URLs |
| **Web Origins** | `https://your-domain.com` | CORS configuration |
| **Direct Access Grants** | Enabled (dev), Disabled (prod) | Password grant for testing |

## 4.2 OAuth 2.0 / OpenID Connect Integration

Kamiwaza supports standard OIDC authentication flows.

**Environment Configuration:**

```
# Keycloak OIDC Settings
AUTH_GATEWAY_KEYCLOAK_URL=https://auth.yourdomain.com
AUTH_GATEWAY_KEYCLOAK_REALM=kamiwaza
AUTH_GATEWAY_KEYCLOAK_CLIENT_ID=kamiwaza-platform

# JWT Validation
AUTH_GATEWAY_JWT_ISSUER=https://auth.yourdomain.com/realms/kamiwaza
AUTH_GATEWAY_JWT_AUDIENCE=kamiwaza-platform
AUTH_GATEWAY_JWKS_URL=https://auth.yourdomain.com/realms/kamiwaza/protocol/openid-
connect/certs
```

**OIDC Discovery Endpoint:**

```
https://auth.yourdomain.com/realms/kamiwaza/.well-known/openid-configuration
```

## 4.3 SAML Integration

**Configure SAML Identity Provider in Keycloak:**

1. Navigate to **Identity Providers** in Keycloak admin console
2. Select **SAML v2.0**
3. Configure SAML settings:
   - **Single Sign-On Service URL**: Your IdP's SSO endpoint
   - **Single Logout Service URL**: Your IdP's logout endpoint
   - **NameID Policy Format**: `urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress`
   - **Principal Type**: Subject NameID
4. Upload IdP metadata XML or configure manually
5. Map SAML attributes to Keycloak user attributes
6. Enable identity provider in login flow

**Attribute Mapping Example:**

```
SAML Attribute        → Keycloak Attribute
------------------      --------------------
email                 → email
firstName             → firstName
lastName              → lastName
memberOf              → roles
```

## 4.4 LDAP / Active Directory Integration

**Configure LDAP Federation:**

1. Navigate to **User Federation** → **Add provider** → **ldap**
2. Configure connection settings:
   - **Connection URL**: `ldap://ldap.company.com:389` or `ldaps://` for SSL
   - **Bind DN**: `cn=admin,dc=company,dc=com`
   - **Bind Credential**: LDAP admin password
3. Configure LDAP search settings:
   - **Users DN**: `ou=users,dc=company,dc=com`
   - **User Object Classes**: `inetOrgPerson, organizationalPerson`
   - **Username LDAP attribute**: `uid` or `sAMAccountName` (AD)
   - **RDN LDAP attribute**: `uid` or `cn`
   - **UUID LDAP attribute**: `entryUUID` or `objectGUID` (AD)
4. Save and test connection
5. Synchronize users: **Synchronize all users** button

**Active Directory Specific Settings:**

- **Vendor**: Active Directory
- **Username LDAP attribute**: `sAMAccountName`
- **RDN LDAP attribute**: `cn`
- **UUID LDAP attribute**: `objectGUID`
- **User Object Classes**: `person, organizationalPerson, user`

**Role Mapping from LDAP Groups:**

1. Go to **Mappers** tab in LDAP federation
2. Create new mapper: **group-ldap-mapper**

- **Mapper Type**: `group-ldap-mapper`
- **LDAP Groups DN**: `ou=groups,dc=company,dc=com`
- **Group Name LDAP Attribute**: `cn`
- **Group Object Classes**: `groupOfNames`
- **Membership LDAP Attribute**: `member`
- **Mode**: `READ_ONLY` or `LDAP_ONLY`

3. Map LDAP groups to Keycloak roles in **Role Mappings**

### 4.5 Single Sign-On (SSO) Setup

**Google SSO Integration:**

1. Create OAuth 2.0 credentials in Google Cloud Console (https://console.cloud.google.com/apis/credentials)
2. Configure authorized redirect URI:

```
https://auth.yourdomain.com/realms/kamiwaza/broker/google/endpoint
```

3. In Keycloak, navigate to **Identity Providers** → **Google**
4. Enter **Client ID** and **Client Secret** from Google Console
5. Save and enable

**Environment Configuration:**

```
# Google SSO
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-client-secret
```

**Microsoft Azure AD / Office 365:**

1. Register application in Azure Portal (https://portal.azure.com)
2. Configure redirect URI:
   `https://auth.yourdomain.com/realms/kamiwaza/broker/oidc/endpoint`
3. In Keycloak, add **OpenID Connect v1.0** provider
4. Configure with Azure AD settings:

- **Authorization URL**: `https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize`
- **Token URL**: `https://login.microsoftonline.com/{tenant}/oauth2/v2.0/token`
- **Client ID**: Azure application ID
- **Client Secret**: Azure client secret

**Testing SSO:**

1. Navigate to Kamiwaza login page
2. Click SSO provider button (Google, Azure, etc.)
3. Authenticate with external identity provider
4. First-time users automatically create Keycloak account
5. Subsequent logins use existing account

---

## 5. Security Configuration

### 5.1 JWT Token Configuration

**Token Security Settings:**

```
# JWT Validation (in env.sh)
AUTH_GATEWAY_JWT_AUDIENCE=kamiwaza-platform  # Required audience claim
AUTH_GATEWAY_JWT_ISSUER=https://auth.yourdomain.com/realms/kamiwaza
AUTH_GATEWAY_JWKS_URL=https://auth.yourdomain.com/realms/kamiwaza/protocol/openid-
connect/certs

# Security Hardening
AUTH_REQUIRE_SUB=true  # Require 'sub' claim (user ID) in tokens
AUTH_EXPOSE_TOKEN_HEADER=false  # Don't expose tokens in response headers
(production)
AUTH_ALLOW_UNSIGNED_STATE=false  # Require signed OIDC state parameter
(production)
```

**Token Algorithms:**

- **Supported**: RS256 (RSA with SHA-256) - asymmetric cryptography
- **Not Supported**: HS256, ES256, or other algorithms

## 5.2 Session Management

**Access Token Expiration:**

Configure in Keycloak: **Realm Settings** → **Tokens**

- **Access Token Lifespan**: 1 hour (default), 5-15 minutes (high security)
- **Refresh Token Lifespan**: 30 days (default)
- **SSO Session Idle**: 30 minutes
- **SSO Session Max**: 10 hours

**Session Timeout Configuration:**

```
# In env.sh
AUTH_GATEWAY_TOKEN_LEEWAY=30  # Clock skew tolerance (seconds)
AUTH_GATEWAY_JWKS_CACHE_TTL=300  # JWKS cache duration (5 minutes)
```

**Best Practices:**

- Short-lived access tokens (5-15 minutes) for high-security environments
- Longer refresh tokens (days) for user convenience
- Implement token refresh in client applications
- Use secure, HTTP-only cookies for browser sessions

## 5.3 HTTPS Enforcement

**Production HTTPS Requirements:**

Kamiwaza enforces HTTPS in production and CI environments when `CI=true` or `KAMIWAZA_ENV=production`.

**TLS Configuration:**

1. Obtain SSL/TLS certificates (Let's Encrypt, commercial CA, etc.)
2. Configure Traefik with TLS:

```
# traefik-dynamic.yml
tls:
  certificates:
    - certFile: /certs/your-domain.crt
      keyFile: /certs/your-domain.key
  options:
    default:
      minVersion: VersionTLS12
      cipherSuites:
        - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
```

3. Update environment:

```
AUTH_GATEWAY_KEYCLOAK_URL=https://auth.yourdomain.com
KAMIWAZA_HTTPS=true
```

## 5.4 Rate Limiting (Optional - Requires Redis)

Rate limiting requires Redis configuration:

```
# Redis connection for rate limiting
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_DB=0
```

**Rate Limit Configuration:**

```
# In auth_gateway_policy.yaml
rate_limits:
  - path: "/api/models*"
    requests_per_minute: 100
    per_user: true

  - path: "/api/auth/token"
    requests_per_minute: 10
    per_ip: true
```

# 6. Monitoring & Troubleshooting

## 6.1 Health Checks

**Auth Service Health Endpoint:**

```
curl http://localhost:7777/health
```

**Response:**

```
{
  "status": "healthy",
  "version": "1.0.0",
  "uptime": 3600.5,
  "KAMIWAZA_USE_AUTH": true,
  "jwks_cache_status": "healthy"
}
```

**Keycloak Health Check:**

```
curl http://localhost:8080/health/ready
```

## 6.2 Log Monitoring

**Auth Service Logs:**

```
# Docker deployments
docker logs kamiwaza-api -f | grep AUTH

# Host deployments
tail -f $KAMIWAZA_LOG_DIR/kamiwaza.log | grep AUTH
```

**Important Log Events:**

- `AUTH_FAILED` - Authentication failure with reason

- `ACCESS_DENIED` - Authorization denial with path/method/roles
- `JWKS_REFRESHED` - JWKS key cache refresh
- `POLICY_RELOADED` - RBAC policy file reload
- `TOKEN_VALIDATED` - Successful token validation

**Keycloak Logs:**

```
docker logs kamiwaza-keycloak -f
```

## 6.3 Common Issues and Solutions

### Issue: 401 Unauthorized on All Requests

**Symptoms:** All API requests return 401 even with valid tokens

**Troubleshooting:**

1. **Check if auth is enabled:**

   ```
   echo $KAMIWAZA_USE_AUTH  # Should be 'true'
   ```

2. **Verify Keycloak is running:**

   ```
   docker ps | grep keycloak
   curl http://localhost:8080/health/ready
   ```

3. **Check JWT issuer matches:**

   ```
   # Decode your token
   echo $TOKEN | cut -d. -f2 | base64 -d | jq .iss

   # Compare with configuration
   echo $AUTH_GATEWAY_JWT_ISSUER
   ```

4. **Verify JWKS endpoint is accessible:**

```
curl $AUTH_GATEWAY_JWKS_URL
```

**Solution:**

- Ensure `AUTH_GATEWAY_JWT_ISSUER` matches token issuer exactly
- Verify Keycloak realm name is correct
- Check network connectivity to Keycloak

### Issue: 403 Forbidden (Valid Token)

**Symptoms:** Token is valid but access denied

**Troubleshooting:**

1. **Check user roles in token:**

```
echo $TOKEN | cut -d. -f2 | base64 -d | jq .realm_access.roles
```

2. **Verify RBAC policy allows access:**

```
cat $KAMIWAZA_ROOT/config/auth_gateway_policy.yaml
```

3. **Check policy file syntax:**

```
# Invalid YAML prevents policy reload
yamllint $KAMIWAZA_ROOT/config/auth_gateway_policy.yaml
```

**Solution:**

- Add required roles to user in Keycloak
- Update RBAC policy to allow endpoint/method/role combination
- Fix YAML syntax errors and reload policy

### Issue: Token Expired Too Quickly

**Symptoms:** Tokens expire after minutes instead of expected duration

**Troubleshooting:**

1. **Check token lifespan in Keycloak:**
   - Navigate to **Realm Settings** → **Tokens**
   - Verify **Access Token Lifespan** setting
2. **Check token claims:**

```
echo $TOKEN | cut -d. -f2 | base64 -d | jq '.exp - .iat'
# Result is token lifetime in seconds
```

**Solution:**

- Increase **Access Token Lifespan** in Keycloak (for development)
- Implement token refresh in client applications
- Use refresh tokens for long-lived sessions

### Issue: Google/SSO Login Not Working

**Symptoms:** SSO redirect fails or returns error

**Troubleshooting:**

1. **Check redirect URI configuration:**
   - Verify redirect URI in Google/Azure console matches Keycloak exactly
   - Format: `https://auth.yourdomain.com/realms/kamiwaza/broker/{provider}/endpoint`
2. **Verify client secret is set:**

```
echo $GOOGLE_CLIENT_SECRET  # Should not be empty
```

3. **Check Keycloak identity provider logs:**

```
docker logs kamiwaza-keycloak -f | grep -i broker
```

**Solution:**

- Update authorized redirect URIs in OAuth provider console
- Ensure client secret is configured in Keycloak
- Enable identity provider in Keycloak authentication flow

## 6.4 Diagnostic Commands

**Test Token Generation:**

```
# Get token from Keycloak
TOKEN=$(curl -s -X POST http://localhost:8080/realms/kamiwaza/protocol/openid-
connect/token \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=password" \
  -d "client_id=kamiwaza-platform" \
  -d "username=testuser" \
  -d "password=testpass" | jq -r .access_token)

# Decode token to inspect claims
echo $TOKEN | cut -d. -f2 | base64 -d | jq .
```

**Test Token Validation:**

```
# Test ForwardAuth validation endpoint directly
curl -v -H "Authorization: Bearer $TOKEN" \
  -H "X-Forwarded-Uri: /api/models" \
  -H "X-Forwarded-Method: GET" \
  http://localhost:7777/auth/validate
```

**Verify JWKS Endpoint:**

```
# Fetch public keys for signature validation
curl http://localhost:8080/realms/kamiwaza/protocol/openid-connect/certs | jq .
```

**Check RBAC Policy:**

```
# View current policy
cat $KAMIWAZA_ROOT/config/auth_gateway_policy.yaml

# Watch for policy reload events
tail -f $KAMIWAZA_LOG_DIR/kamiwaza.log | grep POLICY_RELOADED
```

---

## Appendix A: Environment Variable Reference

### Core Authentication

| Variable | Description | Default |
|---|---|---|
| `KAMIWAZA_USE_AUTH` | Enable/disable authentication | `true` |
| `AUTH_GATEWAY_JWT_ISSUER` | Expected JWT issuer URL | - |
| `AUTH_GATEWAY_JWT_AUDIENCE` | Expected JWT audience claim | - |
| `AUTH_GATEWAY_JWKS_URL` | JWKS endpoint for key fetching | - |
| `AUTH_GATEWAY_POLICY_FILE` | Path to RBAC policy file | `$KAMIWAZA_ROOT/config/auth_gateway_policy` |

## Keycloak Configuration

| Variable | Description | Default | Required |
|---|---|---|---|
| `AUTH_GATEWAY_KEYCLOAK_URL` | Keycloak base URL | `http://localhost:8080` | Yes |
| `AUTH_GATEWAY_KEYCLOAK_REALM` | Keycloak realm name | `kamiwaza` | Yes |
| `AUTH_GATEWAY_KEYCLOAK_CLIENT_ID` | OAuth client ID | `kamiwaza-platform` | Yes |
| `KEYCLOAK_ADMIN_PASSWORD` | Keycloak admin password | `admin` | Yes |

## Security Hardening

| Variable | Description | Default | Required |
|---|---|---|---|
| `AUTH_REQUIRE_SUB` | Require 'sub' claim in tokens | `false` | No |
| `AUTH_EXPOSE_TOKEN_HEADER` | Expose token in response headers | `true` | No |
| `AUTH_ALLOW_UNSIGNED_STATE` | Allow unsigned OIDC state | `true` (dev only) | No |
| `AUTH_GATEWAY_TOKEN_LEEWAY` | Clock skew tolerance (seconds) | `30` | No |
| `AUTH_GATEWAY_JWKS_CACHE_TTL` | JWKS cache duration (seconds) | `300` | No |

## External Identity Providers

| Variable | Description | Default | Required |
|---|---|---|---|
| `GOOGLE_CLIENT_ID` | Google OAuth client ID | - | For Google SSO |
| `GOOGLE_CLIENT_SECRET` | Google OAuth client secret | - | For Google SSO |

# Appendix B: RBAC Policy Examples

## Example 1: Tiered Access by Service

```yaml
version: 1
env: production
default_deny: true

roles:
  - id: admin
    description: "System administrators"
  - id: data_scientist
    description: "Data scientists and ML engineers"
  - id: analyst
    description: "Business analysts and viewers"

endpoints:
  # Model Management - Scientists can create/edit, analysts read-only
  - path: "/api/models*"
    methods: ["GET"]
    roles: ["admin", "data_scientist", "analyst"]

  - path: "/api/models*"
    methods: ["POST", "PUT", "DELETE"]
    roles: ["admin", "data_scientist"]

  # Model Serving - Scientists can deploy, analysts can query
  - path: "/api/serving/deployments*"
    methods: ["GET"]
    roles: ["admin", "data_scientist", "analyst"]

  - path: "/api/serving/deploy"
    methods: ["POST"]
    roles: ["admin", "data_scientist"]

  - path: "/api/serving/generate"
    methods: ["POST"]
    roles: ["admin", "data_scientist", "analyst"]

  # Cluster Management - Admin-only
  - path: "/api/cluster*"
    methods: ["*"]
    roles: ["admin"]

  # Public endpoints
  - path: "/health"
    methods: ["GET"]
    roles: ["*"]
```

### Example 2: Read-Write Separation

```yaml
version: 1
env: production
default_deny: true

roles:
  - id: admin
  - id: editor
  - id: reader

endpoints:
  # Read endpoints - All authenticated users
  - path: "/api/models"
    methods: ["GET"]
    roles: ["admin", "editor", "reader"]

  - path: "/api/vectordb/collections"
    methods: ["GET"]
    roles: ["admin", "editor", "reader"]

  # Write endpoints - Editors and admins only
  - path: "/api/models"
    methods: ["POST", "PUT"]
    roles: ["admin", "editor"]

  - path: "/api/vectordb/collections"
    methods: ["POST", "PUT"]
    roles: ["admin", "editor"]

  # Delete endpoints - Admins only
  - path: "/api/models*"
    methods: ["DELETE"]
    roles: ["admin"]

  - path: "/api/vectordb/collections*"
    methods: ["DELETE"]
    roles: ["admin"]
```

# Other Topics

Content to be added.

# 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, we're here to help:

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

We're committed to making your experience with Kamiwaza as smooth as possible.

## Reporting Issues

When reporting issues to our support team or community, please include:

- **Environment Details**: OS version, Docker version, hardware specs (`bash startup/kamiwazad.sh doctor` or `kamiwaza doctor` for .deb installs is helpful)
- **Error Messages**: Complete error text and stack traces
- **Steps to Reproduce**: Detailed steps that led to the issue
- **Logs**: Relevant log files and container output
- **Configuration**: Any custom configuration or settings

This information helps us provide faster and more accurate solutions to your problems.

# 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

### 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_client'` when using notebooks and Kamiwaza SDK.

**Solution**:

```
!pip uninstall -y kamiwaza
!pip install kamiwaza
```

## 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

# Release Notes

# API Reference

## # Documentation Todo List

## Authentication Service

### login_for_access_token

Login to get access token

### verify_token

Verify authentication token

### create_local_user

Create new local user

### list_users

List all users

### read_users_me

Get current user info

### login_local

Local login

### read_user

Get specific user info

### update_user

Update user details

### delete_user

Delete user

### read_own_permissions

Get own permissions

### create_organization

Create new organization

### read_organization

Get organization info

### update_organization

Update organization

### delete_organization

Delete organization

### create_group

Create new group

### read_group

Get group info

### update_group

Update group

### delete_group

Delete group

### create_role

Create new role

### read_role

Get role info

### update_role

Update role

### delete_role

Delete role

### create_right

Create new right

### read_right

Get right info

### update_right

Update right

### delete_right

Delete right

### add_user_to_group

Add user to group

### remove_user_from_group

Remove user from group

### assign_role_to_group

Assign role to group

### remove_role_from_group

Remove role from group

### assign_right_to_role

Assign right to role

### remove_right_from_role

Remove right from role

## Model Service

### get_model

Get model by ID

### create_model

Create new model

### delete_model

Delete model

### list_models

List all models

### search_models

Search for models

### initiate_model_download

Start model download

### check_download_status

Check model download status

### get_model_files_download_status

Get file download status

### get_model_by_repo_id

Get model by repo ID

### get_model_memory_usage

Get model memory usage

### delete_model_file

Delete model file

### get_model_file

Get model file

### get_model_files_by_model_id

Get files by model ID

### list_model_files

List all model files

### create_model_file

Create model file

### search_hub_model_files

Search hub model files

### get_model_file_memory_usage

Get file memory usage

### create_model_config

Create model config

### get_model_configs

Get model configs

### get_model_configs_for_model

Get configs for model

## Serving Service

### start_ray

Start Ray service

### get_status

Get Ray status

### estimate_model_vram

Estimate model VRAM

### deploy_model

Deploy a model

### list_deployments

List model deployments

### get_deployment

Get deployment info

### stop_deployment

Stop deployment

### get_deployment_status

Get deployment status

### list_model_instances

List model instances

### get_model_instance

Get instance info

### get_health

Get deployment health

### unload_model

Unload model

### load_model

Load model

### simple_generate

Simple text generation

### generate

Advanced text generation

## VectorDB Service

### create_vectordb

Create vector database

### get_vectordbs

List vector databases

### get_vectordb

Get vector database

### remove_vectordb

Remove vector database

### insert_vectors

Insert vectors

### search_vectors

Search vectors

### insert

Simplified vector insertion

### search

Simplified vector search

## Embedding Service

### chunk_text

Chunk text into pieces

### embed_chunks

Generate embeddings

### create_embedding

Create embedding

### get_embedding

Get embedding

### reset_model

Reset embedding model

### call

Generate batch embeddings

### initialize_provider

Initialize embedding provider

### HuggingFaceEmbedding

Create HuggingFace embedder

### get_providers

List available providers

## Retrieval Service

### retrieve_relevant_chunks

Get relevant text chunks

## Ingestion Service

### ingest

Ingest data

### ingest_dataset

Ingest dataset to catalog

### initialize_embedder

Initialize embedder

### process_documents

Process documents

## Cluster Service

### create_location

Create new location

### update_location

Update location

### get_location

Get location info

### list_locations

List locations

### create_cluster

Create new cluster

### get_cluster

Get cluster info

### list_clusters

List clusters

### get_node_by_id

Get node info

### get_running_nodes

List running nodes

### list_nodes

List all nodes

### create_hardware

Create hardware entry

### get_hardware

Get hardware info

### list_hardware

List hardware entries

### get_runtime_config

Get runtime config

### get_hostname

Get cluster hostname

## Lab Service

### list_labs

List all labs

### create_lab

Create new lab

### get_lab

Get lab info

### delete_lab

Delete lab

## Activity Service

### get_recent_activity

Get recent activities

## Catalog Service

### list_datasets

List all datasets

### create_dataset

Create new dataset

### list_containers

List containers

### get_dataset

Get dataset info

### ingest_by_path

Ingest dataset by path

### secret_exists

Check secret existence

### create_secret

Create new secret

## flush_catalog

Clear catalog data

# Activity Service

## Overview

The Activity Service (`ActivityService`) provides comprehensive activity tracking and monitoring functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/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:

```
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

# Authentication Service

## Overview

The Authentication Service ( `AuthService` ) provides comprehensive user authentication and authorization functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/services/auth.py` , this service handles user management, token-based authentication, and role-based access control (RBAC).

## Key Features

- User Authentication (token-based)
- Local User Management
- Organization Management
- Role-Based Access Control (RBAC)
- Group Management
- Permission Management

## Authentication Methods

### Token-Based Authentication

```
# Login to get access token
token = client.auth.login_for_access_token(
    username="user@example.com",
    password="secure_password"
)

# Verify token
user = client.auth.verify_token(authorization="Bearer <token>")
```

### Local Authentication

```
# Create local user
user = client.auth.create_local_user(LocalUserCreate(
    username="newuser",
    email="user@example.com",
    password="securepass"
))

# Login locally
token = client.auth.login_local(username="newuser", password="securepass")
```

## User Management

### Available Methods

- `create_local_user(user: LocalUserCreate) -> User`: Create a new local user
- `list_users() -> List[User]`: List all users in the system
- `read_users_me(authorization: str)`: Get current user's information
- `read_user(user_id: UUID) -> User`: Get specific user information
- `update_user(user_id: UUID, user: UserUpdate) -> User`: Update user details
- `delete_user(user_id: UUID)`: Delete a user
- `read_own_permissions(token: str) -> UserPermissions`: Get current user's permissions

## Organization Management

### Available Methods

- `create_organization(org: OrganizationCreate) -> Organization`: Create new organization
- `read_organization(org_id: UUID) -> Organization`: Get organization details
- `update_organization(org_id: UUID, org: OrganizationUpdate) -> Organization`: Update organization
- `delete_organization(org_id: UUID)`: Delete an organization

## Group Management

### Available Methods

- `create_group(group: GroupCreate) -> Group`: Create new group
- `read_group(group_id: UUID) -> Group`: Get group details
- `update_group(group_id: UUID, group: GroupUpdate) -> Group`: Update group
- `delete_group(group_id: UUID)`: Delete a group
- `add_user_to_group(user_id: UUID, group_id: UUID)`: Add user to group
- `remove_user_from_group(user_id: UUID, group_id: UUID)`: Remove user from group

## Role Management

### Available Methods

- `create_role(role: RoleCreate) -> Role`: Create new role
- `read_role(role_id: UUID) -> Role`: Get role details
- `update_role(role_id: UUID, role: RoleUpdate) -> Role`: Update role
- `delete_role(role_id: UUID)`: Delete a role
- `assign_role_to_group(group_id: UUID, role_id: UUID)`: Assign role to group
- `remove_role_from_group(group_id: UUID, role_id: UUID)`: Remove role from group

## Rights Management

### Available Methods

- `create_right(right: RightCreate) -> Right`: Create new right
- `read_right(right_id: UUID) -> Right`: Get right details
- `update_right(right_id: UUID, right: RightUpdate) -> Right`: Update right
- `delete_right(right_id: UUID)`: Delete a right
- `assign_right_to_role(role_id: UUID, right_id: UUID)`: Assign right to role
- `remove_right_from_role(role_id: UUID, right_id: UUID)`: Remove right from role

## Error Handling

The service includes built-in error handling for common authentication scenarios:

```
try:
    token = client.auth.login_for_access_token(username="user", password="pass")
except AuthenticationError:
    # Handle authentication failures
except APIError as e:
    # Handle API errors
    print(f"Operation failed: {e}")
```

## Best Practices

1. Always use secure passwords and handle credentials securely
2. Implement proper token management (storage and refresh)
3. Use role-based access control for granular permissions
4. Regular audit of user permissions and access rights
5. Clean up unused users, groups, and roles

# Catalog Service

## Overview

The Catalog Service (`CatalogService`) provides comprehensive dataset and container management for the Kamiwaza AI Platform. Located in `kamiwaza_client/services/catalog.py`, this service handles dataset operations, container management, and secret handling for secure data access.

## Key Features

- Dataset Management
- Container Organization
- Secret Management
- Data Ingestion
- Catalog Maintenance

## Dataset Management

### Available Methods

- `list_datasets() -> List[Dataset]`: List all datasets
- `create_dataset(dataset: CreateDataset) -> Dataset`: Create new dataset
- `get_dataset(dataset_id: UUID) -> Dataset`: Get dataset info
- `ingest_by_path(path: str, **kwargs) -> IngestionResponse`: Ingest dataset by path

```python
# List all datasets
datasets = client.catalog.list_datasets()
for dataset in datasets:
    print(f"Dataset: {dataset.name}")
    print(f"Description: {dataset.description}")

# Create new dataset
dataset = client.catalog.create_dataset(CreateDataset(
    name="training-data",
    description="Training dataset for model XYZ",
    metadata={
        "source": "internal",
        "version": "1.0"
    }
))

# Get dataset details
dataset = client.catalog.get_dataset(dataset_id)
print(f"Status: {dataset.status}")
print(f"Size: {dataset.size}")

# Ingest dataset from path
response = client.catalog.ingest_by_path(
    path="/data/training",
    recursive=True,
    file_pattern="*.csv"
)
```

## Container Management

### Available Methods

- `list_containers() -> List[Container]`: List all containers

```python
# List containers
containers = client.catalog.list_containers()
for container in containers:
    print(f"Container: {container.name}")
    print(f"Type: {container.type}")
```

## Secret Management

### Available Methods

- `secret_exists(name: str) -> bool`: Check secret existence
- `create_secret(secret: CreateSecret) -> Secret`: Create new secret
- `flush_catalog() -> None`: Clear catalog data

```
# Check if secret exists
if client.catalog.secret_exists("api-key"):
    print("Secret exists")

# Create new secret
secret = client.catalog.create_secret(CreateSecret(
    name="database-credentials",
    value="secret-value",
    metadata={
        "type": "database",
        "environment": "production"
    }
))

# Clear catalog data
client.catalog.flush_catalog()
```

## Integration with Other Services

The Catalog Service works in conjunction with:

1. Ingestion Service
    - For dataset processing
2. Authentication Service
    - For access control
3. VectorDB Service
    - For vector storage
4. Retrieval Service
    - For data access

## Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    dataset = client.catalog.create_dataset(dataset_config)
except DatasetExistsError:
    print("Dataset already exists")
except StorageError:
    print("Storage operation failed")
except APIError as e:
    print(f"Operation failed: {e}")
```

## Best Practices

1. Use meaningful dataset names
2. Include comprehensive metadata
3. Implement proper error handling
4. Regular catalog maintenance
5. Secure secret management
6. Monitor storage usage
7. Document dataset lineage
8. Validate data before ingestion

## Performance Considerations

- Dataset size impacts ingestion time
- Container organization affects retrieval speed
- Secret management overhead
- Storage capacity requirements
- Catalog operation latency

# Cluster Service

## Overview

The Cluster Service (`ClusterService`) provides comprehensive cluster and infrastructure management for the Kamiwaza AI Platform. Located in `kamiwaza_client/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

# Embedding Service

## Overview

The Embedding Service (`EmbeddingService`) provides comprehensive text embedding functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/services/embedding.py`, this service handles text chunking, embedding generation, and provider management.

## Key Features

- Text Chunking
- Embedding Generation
- Multiple Provider Support
- Batch Processing
- Model Management
- Provider Initialization

## Text Processing

### Available Methods

- `chunk_text(text: str, chunk_size: int = 512) -> List[str]`: Split text into chunks
- `embed_chunks(chunks: List[str]) -> List[List[float]]`: Generate embeddings for chunks
- `create_embedding(text: str) -> List[float]`: Create embedding for text
- `get_embedding(embedding_id: UUID) -> Embedding`: Get existing embedding

```python
# Split text into chunks
chunks = client.embedding.chunk_text(
    text="Long document text...",
    chunk_size=512
)

# Generate embeddings for chunks
embeddings = client.embedding.embed_chunks(chunks)

# Create single embedding
embedding = client.embedding.create_embedding("Sample text")

# Retrieve existing embedding
stored_embedding = client.embedding.get_embedding(embedding_id)
```

## Model Management

### Available Methods

- `reset_model()`: Reset embedding model
- `call(texts: List[str]) -> List[List[float]]`: Generate batch embeddings
- `initialize_provider(provider: str, **kwargs)`: Initialize embedding provider
- `HuggingFaceEmbedding(model_name: str)`: Create HuggingFace embedder
- `get_providers() -> List[str]`: List available providers

```
# Reset model
client.embedding.reset_model()

# Batch embedding generation
embeddings = client.embedding.call(["text1", "text2", "text3"])

# Initialize provider
client.embedding.initialize_provider(
    provider="huggingface",
    model_name="sentence-transformers/all-mpnet-base-v2"
)

# Create HuggingFace embedder
embedder = client.embedding.HuggingFaceEmbedding(
    model_name="sentence-transformers/all-mpnet-base-v2"
)

# List available providers
providers = client.embedding.get_providers()
```

## Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    embedding = client.embedding.create_embedding("text")
except ModelNotFoundError:
    print("Embedding model not found")
except ProviderError as e:
    print(f"Provider error: {e}")
except APIError as e:
    print(f"Operation failed: {e}")
```

## Best Practices

1. Choose appropriate chunk sizes for your use case
2. Use batch processing for better performance
3. Initialize providers with appropriate models
4. Handle model resets properly
5. Monitor embedding quality

6. Use appropriate error handling
7. Consider memory usage for large batches
8. Cache frequently used embeddings

## Provider Configuration

The service supports multiple embedding providers:

1. HuggingFace
    - Supports various model architectures
    - Customizable model selection
    - Local and remote inference
2. Custom Providers
    - Extensible provider interface
    - Custom model integration
    - Provider-specific configurations

# Ingestion Service

## Overview

The Ingestion Service ( `IngestionService` ) provides comprehensive data ingestion functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/services/ingestion.py` , this service handles data ingestion workflows, dataset processing, and document handling with embedding capabilities.

## Key Features

- Data Ingestion
- Dataset Catalog Integration
- Document Processing
- Embedding Generation
- Batch Processing Support

## Data Ingestion

### Available Methods

- `ingest(data: Union[str, List[str], Dict[str, Any]], **kwargs) -> IngestionResponse` : Ingest data
- `ingest_dataset(dataset: Dataset, **kwargs) -> DatasetIngestionResponse` : Ingest dataset to catalog
- `initialize_embedder(provider: str = "default", **kwargs) -> None` : Initialize embedding provider
- `process_documents(documents: List[Document], **kwargs) -> ProcessingResponse` : Process documents

```
# Simple data ingestion
response = client.ingestion.ingest(
    data="Sample text data",
    chunk_size=512
)

# Dataset ingestion
response = client.ingestion.ingest_dataset(
    dataset=dataset_obj,
    embedding_config={
        "provider": "huggingface",
        "model": "sentence-transformers/all-mpnet-base-v2"
    }
)

# Initialize embedder
client.ingestion.initialize_embedder(
    provider="huggingface",
    model_name="sentence-transformers/all-mpnet-base-v2"
)

# Process documents
response = client.ingestion.process_documents(
    documents=[
        Document(text="doc1", metadata={"source": "file1"}),
        Document(text="doc2", metadata={"source": "file2"})
    ],
    chunk_size=512,
    overlap=50
)
```

## Integration with Other Services

The Ingestion Service works in conjunction with:

1. Embedding Service
   - For generating embeddings of ingested text
2. VectorDB Service
   - For storing processed vectors
3. Catalog Service
   - For dataset management
4. Retrieval Service
   - For accessing processed documents

## Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    response = client.ingestion.ingest(data)
except EmbeddingError:
    print("Embedding generation failed")
except VectorDBError:
    print("Vector storage failed")
except ProcessingError as e:
    print(f"Document processing failed: {e}")
except APIError as e:
    print(f"Operation failed: {e}")
```

## Best Practices

1. Initialize embedder before ingestion
2. Use appropriate chunk sizes
3. Include relevant metadata
4. Process documents in batches
5. Monitor ingestion progress
6. Handle errors appropriately
7. Clean up failed ingestions
8. Validate data before ingestion

## Performance Considerations

- Batch size affects processing speed
- Embedding generation time
- Vector database insertion overhead
- Memory usage during processing
- Network bandwidth for large datasets

## Data Formats

The service supports various input formats:

1. Raw Text
    - Single strings
    - Lists of strings
2. Structured Data
    - JSON objects
    - Dictionaries
3. Documents
    - Custom Document objects
    - Metadata support
4. Datasets
    - Catalog integration
    - Batch processing

# Lab Service

## Overview

The Lab Service (`LabService`) provides comprehensive lab environment management for the Kamiwaza AI Platform. Located in `kamiwaza_client/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:

```
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

# Model Service

## Overview

The Model Service (`ModelService`) provides comprehensive model management functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/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

# Retrieval Service

## Overview

The Retrieval Service (`RetrievalService`) provides text chunk retrieval functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/services/retrieval.py`, this service handles the retrieval of relevant text chunks based on queries, enabling efficient information retrieval.

## Key Features

- Relevant Text Chunk Retrieval
- Query-based Search
- Integration with Vector Database
- Semantic Search Capabilities

## Text Chunk Retrieval

### Available Methods

- `retrieve_relevant_chunks(query: str, k: int = 5) -> List[TextChunk]`: Get relevant text chunks based on query

```
# Retrieve relevant chunks
chunks = client.retrieval.retrieve_relevant_chunks(
    query="What is machine learning?",
    k=5  # Number of chunks to retrieve
)

# Process retrieved chunks
for chunk in chunks:
    print(f"Text: {chunk.text}")
    print(f"Score: {chunk.score}")
    print(f"Source: {chunk.metadata.get('source')}")
```

## Integration with Other Services

The Retrieval Service works in conjunction with:

1. Embedding Service
    - For converting queries into vector representations
2. VectorDB Service
    - For performing similarity search
3. Ingestion Service
    - For accessing processed and stored text chunks

## Error Handling

The service includes built-in error handling for common scenarios:

```
try:
    chunks = client.retrieval.retrieve_relevant_chunks(
        query="example query"
    )
except VectorDBError:
    print("Vector database error")
except EmbeddingError:
    print("Embedding generation error")
except APIError as e:
    print(f"Operation failed: {e}")
```

## Best Practices

1. Use specific and focused queries
2. Adjust the number of chunks (k) based on your needs
3. Consider chunk relevance scores
4. Process chunks in order of relevance
5. Handle empty result sets appropriately
6. Implement proper error handling
7. Consider caching frequently retrieved chunks

8. Monitor retrieval performance

## Performance Considerations

- Query length affects retrieval time
- Number of chunks (k) impacts response time
- Vector database size influences search speed
- Embedding generation adds processing overhead

# Serving Service

## Overview

The Serving Service (`ServingService`) provides comprehensive model deployment and serving capabilities for the Kamiwaza AI Platform. Located in `kamiwaza_client/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(deployment: CreateModelDeployment) -> ModelDeployment`: Deploy a model
- `list_deployments() -> List[ModelDeployment]`: List all deployments
- `list_active_deployments() -> List[UIModelDeployment]`: List only active deployments with running instances
- `get_deployment(deployment_id: UUID) -> ModelDeployment`: Get deployment details
- `stop_deployment(deployment_id: UUID)`: Stop a deployment
- `get_deployment_status(deployment_id: UUID) -> DeploymentStatus`: Get deployment status

```
# Estimate VRAM requirements
vram_needed = client.serving.estimate_model_vram(model_id)

# Deploy a model
deployment = client.serving.deploy_model(CreateModelDeployment(
    model_id=model_id,
    name="my-deployment",
    replicas=1,
    max_concurrent_requests=4
))

# 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. http://hostname:port/v1)

# Get deployment status
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:

```
try:
    deployment = client.serving.deploy_model(deployment_config)
except DeploymentError as e:
    print(f"Deployment failed: {e}")
except ResourceError as e:
    print(f"Resource allocation failed: {e}")
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

# VectorDB Service

## Overview

The VectorDB Service (`VectorDBService`) provides comprehensive vector database management functionality for the Kamiwaza AI Platform. Located in `kamiwaza_client/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

```python
# 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:

```
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