Red Hat OpenShift AI Self-Managed 3.0

Working with Llama Stack

Working with Llama Stack in Red Hat OpenShift AI Self-Managed

Last Updated: 2025-12-01

Red Hat OpenShift AI Self-Managed 3.0 Working with Llama Stack

Working with Llama Stack in Red Hat OpenShift AI Self-Managed

Legal Notice

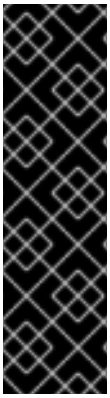
Abstract

As a cluster administrator, you can use the Llama Stack Operator in Red Hat OpenShift AI.

Table of Contents

CHAPTER 1. OVERVIEW OF LLAMA STACK

Llama Stack is a unified AI runtime environment designed to simplify the deployment and management of generative AI workloads on OpenShift AI. Llama Stack integrates LLM inference servers, vector databases, and retrieval services in a single stack, optimized for Retrieval-Augmented Generation (RAG) and agent-based AI workflows. In OpenShift, the Llama Stack Operator manages the deployment lifecycle of these components, ensuring scalability, consistency, and integration with OpenShift AI projects.



IMPORTANT

Llama Stack integration is currently available in Red Hat OpenShift AI 3.0 as a Technology Preview feature. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Llama Stack includes the following components:

- **Inference model servers** such as vLLM, designed to efficiently serve large language models.
- **Vector storage** solutions, primarily Milvus, to store embeddings generated from your domain data.
- **Retrieval and embedding management** workflows using integrated tools, such as Docling, to handle continuous data ingestion and synchronization.
- **Integration with OpenShift AI** by using the **LlamaStackDistribution** custom resource, simplifying configuration and deployment.

For information about how to deploy Llama Stack in OpenShift AI, see Deploying a RAG stack in a project.

NOTE

The Llama Stack Operator is not currently supported on IBM Power/Z machines.

Additional resources

- Llama Stack Demos repository
- Llama Stack Kubernetes Operator documentation
- Llama Stack documentation

1.1. LLAMA STACK APIS

You can use the following APIs from Llama Stack for AI actions such as evaluation, scoring, and inference:

1.1.1. Supported Llama Stack APIs in OpenShift AI

1.1.1.1. Agents API

- **Endpoint: /v1alpha/agents**.
- **Providers:** All agent backends deployed through OpenShift AI.
- **Support level:** Developer Preview.

The Agents API allows you to create and manage AI agents.

1.1.1.2. Datasets_IO API

- **Endpoint: /v1beta/datasetio**.
- **Providers:** All dataset_io backends deployed through OpenShift AI.
- **Support level:** Technology Preview.

The Dataset_IO API manages the input and output of datasets and their content.

1.1.1.3. Evaluation API

- **Endpoint: /v1beta/eval**.
- **Providers:** All evaluation backends deployed through OpenShift AI.
- **Support level:** Developer Preview.

The Evaluation API defines an evaluation task for models and datasets

1.1.1.4. Inference API

- **Endpoint: /v1alpha/inference**.
- **Providers:** All inference backends deployed through OpenShift AI.
- **Support level:** Developer Preview.

WARNING

The majority of the Inference API is deprecated. The Inference providers use the Completions and Chat Completions APIs now.

The Inference API enables conversational, message-based interactions with models served by Llama Stack in OpenShift AI.

1.1.1.5. Safety API

- **Endpoint:** /**v1**/**safety**.
- **Providers:** All safety backends deployed through OpenShift AI.
- **Support level:** Technology Preview.

The Safety API detects and prevents harmful content in model inputs and outputs.

1.1.1.6. Tool Runtime API

- **Endpoint:** /**v1**/**tool-runtime**.
- **Providers:** All tool runtime backends deployed through OpenShift AI.
- **Support level:** Developer Preview.

The Tool Runtime API allows a model to dynamically call a tool at runtime.

1.1.1.7. Vector_IO API

- **Endpoint:** /**v1**/**vector-io**.
- **Providers:** All vector_io backends deployed through OpenShift AI.
- **Support level:** Developer Preview.

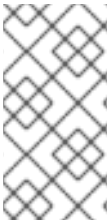
The Vector_IO API allows you to manage and query vector embeddings: numeric representations of data.

1.2. OPENAI-COMPATIBLE APIS IN LLAMA STACK

OpenShift AI includes a Llama Stack component that exposes OpenAI-compatible APIs. These APIs enable you to reuse existing OpenAI SDKs, tools, and workflows directly within your OpenShift environment, without changing your client code. This compatibility layer supports retrieval-augmented generation (RAG), inference, and embedding workloads by using the same endpoints, schemas, and authentication model as OpenAI.

This compatibility layer has the following capabilities:

- **Standardized endpoints**: REST API paths align with OpenAI specifications.
- **Schema parity:** Request and response fields follow OpenAI data structures.



NOTE

When connecting OpenAI SDKs or third-party tools to OpenShift AI, you must update the client configuration to use your deployment's Llama Stack route as the **base_url**. This ensures that API calls are sent to the OpenAI-compatible endpoints that run inside your OpenShift cluster, rather than to the public OpenAI service.

1.2.1. Supported OpenAI-compatible APIs in OpenShift AI

1.2.1.1. Chat Completions API

- **Endpoint: /v1/openai/v1/chat/completions**.
- **Providers:** All inference back ends deployed through OpenShift AI.
- **Support level:** Technology Preview.

The Chat Completions API enables conversational, message-based interactions with models served by Llama Stack in OpenShift AI.

1.2.1.2. Completions API

- **Endpoint: /v1/openai/v1/completions**.
- **Providers:** All inference backends managed by OpenShift AI.
- **Support level:** Technology Preview.

The Completions API supports single-turn text generation and prompt completion.

1.2.1.3. Embeddings API

- **Endpoint: /v1/openai/v1/embeddings**.
- **Providers:** All embedding models enabled in OpenShift AI.

The Embeddings API generates numerical embeddings for text or documents that can be used in downstream semantic search or RAG applications.

1.2.1.4. Files API

- **Endpoint: /v1/openai/v1/files**.
- **Providers:** File system-based file storage provider for managing files and documents stored locally in your cluster.
- **Support level:** Technology Preview.

The Files API manages file uploads for use in embedding and retrieval workflows.

1.2.1.5. Vector Stores API

- **Endpoint: /v1/openai/v1/vector_stores/**.
- **Providers:** Inline and Remote Milvus configured in OpenShift AI.
- **Support level:** Technology Preview.

The Vector Stores API manages the creation, configuration, and lifecycle of vector store resources in Llama Stack. Through this API, you can create new vector stores, list existing ones, delete unused stores, and query their metadata, all using OpenAI-compatible request and response formats.

1.2.1.6. Vector Store Files API

- **Endpoint: /v1/openai/v1/vector_stores/{vector_store_id}/files**.

- **Providers:** Local inline provider configured for file storage and retrieval.
- **Support level:** Developer Preview.

The Vector Store Files API implements the OpenAI Vector Store Files interface and manages the link between document files and Milvus vector stores used for RAG.

1.2.1.7. Models API

- **Endpoint: /v1/openai/v1/models**.
- **Providers:** All model-serving back ends configured within OpenShift AI.
- **Support level:** Technology Preview.

The Models API lists and retrieves available model resources from the Llama Stack deployment running on OpenShift AI. By using the Models API, you can enumerate models, view their capabilities, and verify deployment status through a standardized OpenAI-compatible interface.

1.2.1.8. Responses API

- **Endpoint: /v1/openai/v1/responses**.
- **Providers:** All agents, inference and vector providers configured in OpenShift AI.
- **Support level:** Developer Preview.

The Responses API generates model outputs by combining inference, file search, and tool-calling capabilities through a single OpenAI-compatible endpoint. It is particularly useful for retrieval-augmented generation (RAG) workflows that rely on the **file_search** tool to retrieve context from vector stores.

NOTE

The Responses API is an experimental feature that is still under active development in OpenShift AI. While the API is already functional and suitable for evaluation, some endpoints and parameters remain under implementation and might change in future releases. This API is provided for testing and feedback purposes only and is not recommended for production use.

Additional resources

- OpenAI API Compatibility

1.2.2. OpenAI compatibility for RAG APIs in Llama Stack

OpenShift AI supports OpenAI-compatible request and response schemas for Llama Stack RAG workflows. You can use OpenAI clients and schemas for files, vector stores, and Responses API file search end-to-end.

OpenAI compatibility enables the following capabilities:

- You can use OpenAI SDKs and tools with Llama Stack by setting the client **base_url** to the Llama Stack OpenAI path, **/v1/openai/v1**.

- You can manage files and vector stores by using OpenAI-compatible endpoints. You can then invoke RAG workflows by using the Responses API with the **file_search** tool.

Additional resources

- OpenAI API Compatibility
- OpenAI API Reference
- llama-stack-client-python

1.3. LLAMA STACK API PROVIDER SUPPORT

You can use Llama Stack to enable various Provider APIs and providers in OpenShift AI. The following table lists the supported providers included in OpenShift AI

WARNING

The support status of the Llama Stack API providers has shifted between Technology Preview and Developer Preview across OpenShift AI versions.

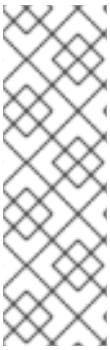
Provider API	Providers	How to Enable	Disconnected support	Support status
Agents	**inline::meta-reference** **NOTE** The Responses API is accessible from the Agents provider API.	Enabled by default	Yes	Developer Preview
Dataset_IO	**inline::localfs**	Enabled by default	Yes	Technology Preview
	remote::huggingface	Enabled by default	No	Technology Preview

Provider API	Providers	How to Enable	Disconnected support	Support status
Evaluation	**inline::ragas**	Set the **EMBEDDING_MODEL** environment variable	No	Technology Preview
	remote::lmeval	Enabled by default	No	Technology Preview
	remote::ragas	See the "Configuring the Ragas remote provider for production" documentation	No	Technology Preview
Files	**inline::localfs**	Enabled by default	No	Technology Preview
Inference	**inline::sentence-transformers**	Enabled by default	Yes	Technology Preview
	remote::vllm	Set the **VLLM_URL** environment variable	Yes	Technology Preview
	remote::azure	Set the **AZURE_API_KEY** environment variable	No	Technology Preview
	remote::bedrock	Set the **AWS_ACCESS_KEY_ID** environment variable	No	Technology Preview
	remote::openai	Set the **OPENAI_API_KEY** environment variable	No	Technology Preview
	remote::vertexai	Set the **VERTEX_AI_PROJECT** environment variable	No	Technology Preview
	remote::watsonx	Set the **WATSONX_API_KEY** environment variable	No	Technology Preview
Safety	**remote::trustyai_fms**	Enabled by default	No	Technology Preview
Scoring	**inline::basic**	Enabled by default	No	Technology Preview

Provider API	Providers	How to Enable	Disconnected support	Support status
	inline::braintrust	Enabled by default	No	Technology Preview
	inline::llm-as-a-judge	Enabled by default	No	Technology Preview
Tool_Runtime	**inline::rag-runtime**	Enabled by default	No	Developer Preview
	remote::brave-search	Enabled by default	No	Developer Preview
	remote::model-context-protocol	Enabled by default	No	Developer Preview
	remote::tavily-search	Enabled by default	No	Developer Preview
Vector_IO	**inline::faiss**	Set the **ENABLE_FAISS** environment variable	No	Technology Preview
	inline::milvus	Enabled by default	Yes	Technology Preview
	remote::milvus	Set the **MILVUS_ENDPOINT** environment variable	Yes	Technology Preview

CHAPTER 2. ACTIVATING THE LLAMA STACK OPERATOR

You can activate the Llama Stack Operator on your OpenShift cluster by setting its **managementState** to **Managed** in the OpenShift AI Operator **DataScienceCluster** custom resource (CR). This setting enables Llama-based model serving without reinstalling or directly editing Operator subscriptions. You can edit the CR in the OpenShift web console or by using the OpenShift CLI (**oc**).



NOTE

As an alternative to following the steps in this procedure, you can activate the Llama Stack Operator from the OpenShift CLI (**oc**) by running the following command:

```
$ oc patch datasciencecluster <name> --type=merge -p {"spec":{"components":
{"llamastackoperator":{"managementState":"Managed"}}}}
```

Replace *<name>* with your **DataScienceCluster** name, for example, **default-dsc**.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have cluster administrator privileges.
- You have installed the OpenShift CLI (**oc**) as described in the appropriate documentation for your cluster:
  - Installing the OpenShift CLI for OpenShift Container Platform
  - Installing the OpenShift CLI for Red Hat OpenShift Service on AWS
- You have installed the Red Hat OpenShift AI Operator on your cluster.
- You have a **DataScienceCluster** custom resource in your environment; the default is **default-dsc**.
- Your infrastructure supports GPU-enabled instance types, for example, **g4dn.xlarge** on AWS.
- You have enabled GPU support in OpenShift AI, including installing the Node Feature Discovery Operator and NVIDIA GPU Operator. For more information, see Installing the Node Feature Discovery Operator and Enabling NVIDIA GPUs.
- You have created a **NodeFeatureDiscovery** resource instance on your cluster, as described in Installing the Node Feature Discovery Operator and creating a NodeFeatureDiscovery instance in the NVIDIA documentation.
- You have created a **ClusterPolicy** resource instance with default values on your cluster, as described in Creating the ClusterPolicy instance in the NVIDIA documentation.

Procedure

1. Log in to the OpenShift web console as a cluster administrator.
2. In the **Administrator** perspective, click **Operators → Installed Operators**.
3. Click the **Red Hat OpenShift AI Operator** to open its details.

4. Click the **Data Science Cluster** tab.
5. On the **DataScienceClusters** page, click the **default-dsc** object.
6. Click the **YAML** tab.
   An embedded YAML editor opens, displaying the configuration for the **DataScienceCluster** custom resource.
7. In the YAML editor, locate the **spec.components** section. If the **llamastackoperator** field does not exist, add it. Then, set the **managementState** field to **Managed**:

```
spec:
  components:
    llamastackoperator:
      managementState: Managed
```

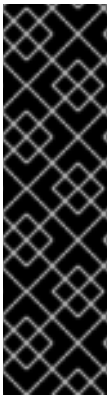
8. Click **Save** to apply your changes.

Verification

After you activate the Llama Stack Operator, verify that it is running in your cluster:

1. In the OpenShift web console, click **Workloads → Pods**.
2. From the **Project** list, select the **redhat-ods-applications** namespace.
3. Confirm that a pod with the label **app.kubernetes.io/name=llama-stack-operator** is displayed and has a status of **Running**.

CHAPTER 3. DEPLOYING A RAG STACK IN A PROJECT



IMPORTANT

This feature is currently available in Red Hat OpenShift AI 3.0 as a Technology Preview feature. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

As an OpenShift cluster administrator, you can deploy a Retrieval-Augmented Generation (RAG) stack in OpenShift AI. This stack provides the infrastructure, including LLM inference, vector storage, and retrieval services that data scientists and AI engineers use to build conversational workflows in their projects.

To deploy the RAG stack in a project, complete the following tasks:

- Activate the Llama Stack Operator in OpenShift AI.
- Enable GPU support on the OpenShift cluster. This task includes installing the required NVIDIA Operators.
- Deploy an inference model, for example, the llama-3.2-3b-instruct model. This task includes creating a storage connection and configuring GPU allocation.
- Create a **LlamaStackDistribution** instance to enable RAG functionality. This action deploys LlamaStack alongside a Milvus vector store and connects both components to the inference model.
- Ingest domain data into Milvus by running Docling in an AI pipeline or Jupyter notebook. This process keeps the embeddings synchronized with the source data.
- Expose and secure the model endpoints.

3.1. OVERVIEW OF RAG

Retrieval-augmented generation (RAG) in OpenShift AI enhances large language models (LLMs) by integrating domain-specific data sources directly into the model's context. Domain-specific data sources can be structured data, such as relational database tables, or unstructured data, such as PDF documents.

RAG indexes content and builds an embedding store that data scientists and AI engineers can query. When data scientists or AI engineers pose a question to a RAG chatbot, the RAG pipeline retrieves the most relevant pieces of data, passes them to the LLM as context, and generates a response that reflects both the prompt and the retrieved content.

By implementing RAG, data scientists and AI engineers can obtain tailored, accurate, and verifiable answers to complex queries based on their own datasets within a project.

3.1.1. Audience for RAG

The target audience for RAG is practitioners who build data-grounded conversational AI applications using OpenShift AI infrastructure.

For Data Scientists

Data scientists can use RAG to prototype and validate models that answer natural-language queries against data sources without managing low-level embedding pipelines or vector stores. They can focus on creating prompts and evaluating model outputs instead of building retrieval infrastructure.

For MLOps Engineers

MLOps engineers typically deploy and operate RAG pipelines in production. Within OpenShift AI, they manage LLM endpoints, monitor performance, and ensure that both retrieval and generation scale reliably. RAG decouples vector store maintenance from the serving layer, enabling MLOps engineers to apply CI/CD workflows to data ingestion and model deployment alike.

For Data Engineers

Data engineers build workflows to load data into storage that OpenShift AI indexes. They keep embeddings in sync with source systems, such as S3 buckets or relational tables to ensure that chatbot responses are accurate.

For AI Engineers

AI engineers architect RAG chatbots by defining prompt templates, retrieval methods, and fallback logic. They configure agents and add domain-specific tools, such as OpenShift job triggers, enabling rapid iteration.

3.2. OVERVIEW OF VECTOR DATABASES

Vector databases are a crucial component of retrieval-augmented generation (RAG) in OpenShift AI. They store and index vector embeddings that represent the semantic meaning of text or other data. When you integrate vector databases with Llama Stack in OpenShift AI, you can build RAG applications that combine large language models (LLMs) with relevant, domain-specific knowledge.

Vector databases provide the following capabilities:

- Store vector embeddings generated by embedding models.
- Support efficient similarity search to retrieve semantically related content.
- Enable RAG workflows by supplying the LLM with contextually relevant data from a specific domain.

When you deploy RAG workloads in OpenShift AI, you can deploy vector databases through the Llama Stack Operator. Currently, OpenShift AI supports the following vector databases:

- **Inline Milvus Lite** An Inline Milvus vector database runs embedded within the Llama Stack Distribution (LSD) pod and is suitable for lightweight experimentation and small-scale development. Inline Milvus stores data in a local SQLite database and is limited in scale and persistence.
- **Inline FAISS** Inline FAISS provides an alternative lightweight vector store for RAG workloads. FAISS (Facebook AI Similarity Search) is an open-source library for efficient similarity search and clustering of dense vectors. When configured inline with a SQLite backend, FAISS runs entirely within the Llama Stack container and stores embeddings locally without requiring a separate database service. Inline FAISS is ideal for testing and experimental RAG deployments.
- **Remote Milvus** A remote Milvus vector database runs as a standalone service in your project namespace or as an external managed deployment. Remote Milvus is best for production-grade RAG use cases because it provides persistence, scalability, and isolation from the Llama Stack

Distribution (LSD) pod. In OpenShift environments, you must deploy Milvus with an etcd service directly in your project. For more information on using etcd services, see Providing redundancy with etcd.

Consider the following points when you decide which vector database to use for your RAG workloads:

- Use **inline Milvus Lite** if you want to experiment quickly with RAG in a self-contained setup and do not require persistence across pod restarts.
- Use **inline FAISS** if you need a lightweight, in-process vector store with local persistence through SQLite and no network dependency.
- Use **remote Milvus** if you need reliable storage, high availability, and the ability to scale RAG workloads in your OpenShift AI environment.

3.2.1. Overview of Milvus vector databases

Milvus is an open source vector database designed for high-performance similarity search across embedding data. In OpenShift AI, Milvus is supported as a remote vector database provider for the Llama Stack Operator. Milvus enables retrieval-augmented generation (RAG) workloads that require persistence, scalability, and efficient search across large document collections.

Milvus vector databases provide you with the following capabilities in OpenShift AI:

- Similarity search using Approximate Nearest Neighbor (ANN) algorithms.
- Persistent storage support for vectors.
- Indexing and query optimizations for embedding-based search.
- Integration with external metadata and APIs.

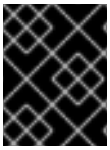
In OpenShift AI, you can use Milvus vector databases in the following operational modes:

- **Inline Milvus Lite**, which runs embedded in the Llama Stack Distribution pod for testing or small-scale experiments.
- **Remote Milvus**, which runs as a standalone service in your OpenShift project or as an external managed Milvus service. Remote Milvus is recommended for production workloads.

When you deploy a remote Milvus vector database, you must run the following components in your OpenShift project:

- **Secret (milvus-secret)**: Stores sensitive data such as the Milvus root password.
- **PersistentVolumeClaim (milvus-pvc)**: Provides persistent storage for Milvus data.
- **Deployment (etcd-deployment)**: Runs an etcd instance that Milvus uses for metadata storage and service coordination.
- **Service (etcd-service)**: Exposes the etcd port for Milvus to connect to.
- **Deployment (milvus-standalone)**: Runs Milvus in standalone mode and connects it to the etcd service and PVC.
- **Service (milvus-service)**: Exposes Milvus gRPC (19530) and HTTP (9091 health check) ports for client access.

Milvus requires an etcd service to manage metadata such as collections, indexes, and partitions, and to provide service discovery and coordination among Milvus components. Even when running in standalone mode, Milvus depends on etcd to operate correctly and maintain metadata consistency. For more information on using etcd services, see Providing redundancy with etcd.



IMPORTANT

Do not use the OpenShift control plane etcd for Milvus. You must deploy a separate etcd instance inside your project or connect to an external etcd service.

Use Remote Milvus when you require a persistent, scalable, and production-ready vector database that integrates seamlessly with OpenShift AI. Consider choosing a remote Milvus vector database if your deployment must cater for the following requirements:

- Persistent vector storage across restarts or upgrades.
- Scalable indexing and high-performance vector search.
- A production-grade RAG architecture integrated with OpenShift AI.

3.2.2. Overview of FAISS vector databases

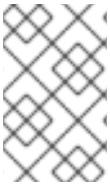
The FAISS (Facebook AI Similarity Search) library is an open-source framework for high-performance vector search and clustering. It is optimized for dense numerical embeddings and supports both CPU and GPU execution. You can enable inline FAISS in OpenShift AI with an embedded SQLite backend in your Llama Stack Distribution. This configures Llama Stack to use FAISS as an in-process vector store, storing embeddings locally within the container without requiring a separate vector database service.

Inline FAISS enables efficient similarity search and retrieval within retrieval augmented generation (RAG) workflows. It operates entirely within the **LlamaStackDistribution** instance, making it a lightweight option for experimental and testing environments.

Inline FAISS offers the following benefits:

- Simplified setup with no external database or network dependencies.
- Persistent local storage of FAISS vector data.
- Reduced latency for embedding ingestion and query operations.
- Compatibility with OpenAI-compatible Vector Store API endpoints.

Inline FAISS stores vectors either in memory or in a local SQLite database file, allowing the deployment to retain vector data across sessions with minimal overhead.



NOTE

Inline FAISS is best for experimental or testing environments. It does not provide distributed storage or high availability. For production-grade workloads that require scalability or redundancy, consider using an external vector database, such as Milvus.

3.3. DEPLOYING A LLAMA MODEL WITH KSERVE

To use Llama Stack and retrieval-augmented generation (RAG) workloads in OpenShift AI, you must deploy a Llama model with a vLLM model server and configure KServe in KServe RawDeployment mode.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have logged in to Red Hat OpenShift AI.
- You have cluster administrator privileges for your OpenShift cluster.
- You have activated the Llama Stack Operator.
- You have installed KServe.
- You have enabled the model serving platform. For more information about enabling the model serving platform, see Enabling the model serving platform.
- You can access the model serving platform in the dashboard configuration. For more information about setting dashboard configuration options, see Customizing the dashboard.
- You have enabled GPU support in OpenShift AI, including installing the Node Feature Discovery Operator and NVIDIA GPU Operator. For more information, see Installing the Node Feature Discovery Operator and Enabling NVIDIA GPUs.
- You have installed the OpenShift CLI (**oc**) as described in the appropriate documentation for your cluster:
  - Installing the OpenShift CLI for OpenShift Container Platform
  - Installing the OpenShift CLI for Red Hat OpenShift Service on AWS
- You have created a project.
- The vLLM serving runtime is installed and available in your environment.
- You have created a storage connection for your model that contains a **URI - v1** connection type. This storage connection must define the location of your Llama 3.2 model artifacts. For example, **oci://quay.io/redhat-ai-services/modelcar-catalog:llama-3.2-3b-instruct**. For more information about creating storage connections, see Adding a connection to your project .

PROCEDURE

These steps are only supported in OpenShift AI versions 2.19 and later.

1. In the OpenShift AI dashboard, navigate to the project details page and click the **Deployments** tab.
2. In the **Model serving platform** tile, click **Select model**.
3. Click the **Deploy model** button.
   The **Deploy model** dialog opens.
4. Configure the deployment properties for your model:
   a. In the **Model deployment name** field, enter a unique name for your deployment.

b. In the **Serving runtime** field, select **vLLM NVIDIA GPU serving runtime for KServe** from the drop-down list.

c. In the **Deployment mode** field, select **KServe RawDeployment** from the drop-down list.

d. Set **Number of model server replicas to deploy** to **1**.

e. In the **Model server size** field, select **Custom** from the drop-down list.

- Set **CPUs requested** to **1 core**.
- Set **Memory requested** to **10 GiB**.
- Set **CPU limit** to **2 core**.
- Set **Memory limit** to **14 GiB**.
- Set **Accelerator** to **NVIDIA GPUs**.
- Set **Accelerator count** to **1**.

f. From the **Connection type**, select a relevant data connection from the drop-down list.

5. In the **Additional serving runtime arguments** field, specify the following recommended arguments:

```
--dtype=half
--max-model-len=20000
--gpu-memory-utilization=0.95
--enable-chunked-prefill
--enable-auto-tool-choice
--tool-call-parser=llama3_json
--chat-template=/app/data/template/tool_chat_template_llama3.2_json.jinja
```

a. Click **Deploy**.

NOTE

Model deployment can take several minutes, especially for the first model that is deployed on the cluster. Initial deployment may take more than 10 minutes while the relevant images download.

Verification

1. Verify that the **kserve-controller-manager** and **odh-model-controller** pods are running:

a. Open a new terminal window.

b. Log in to your OpenShift cluster from the CLI:

c. In the upper-right corner of the OpenShift web console, click your user name and select **Copy login command**.

d. After you have logged in, click **Display token**.

e. Copy the **Log in with this token** command and paste it in the OpenShift CLI (**oc**).

```
$ oc login --token=<token> --server=<openshift_cluster_url>
```

f. Enter the following command to verify that the **kserve-controller-manager** and **odh-model-controller** pods are running:

```
$ oc get pods -n redhat-ods-applications | grep -E 'kserve-controller-manager|odh-model-controller'
```

g. Confirm that you see output similar to the following example:

```
kserve-controller-manager-7c865c9c9f-xyz12   1/1     Running   0          4m21s
odh-model-controller-7b7d5fd9cc-wxy34        1/1     Running   0          3m55s
```

h. If you do not see either of the **kserve-controller-manager** and **odh-model-controller** pods, there could be a problem with your deployment. In addition, if the pods appear in the list, but their **Status** is not set to **Running**, check the pod logs for errors:

```
$ oc logs <pod-name> -n redhat-ods-applications
```

i. Check the status of the inference service:

```
$ oc get inferenceservice -n llamastack
$ oc get pods -n <project name> | grep llama
```

- The deployment automatically creates the following resources:
  - A **ServingRuntime** resource.
  - An **InferenceService** resource, a **Deployment**, a pod, and a service pointing to the pod.
- Verify that the server is running. For example:

```
$ oc logs llama-32-3b-instruct-predictor-77f6574f76-8nl4r  -n <project name>
```

Check for output similar to the following example log:

```
INFO     2025-05-15 11:23:52,750 __main__:498 server: Listening on ['::', '0.0.0.0']:8321
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO     2025-05-15 11:23:52,765 __main__:151 server: Starting up
INFO:     Application startup complete.
INFO:     Uvicorn running on http://['::', '0.0.0.0']:8321 (Press CTRL+C to quit)
```

- The deployed model displays in the **Deployments** tab on the project details page for the project it was deployed under.

2. If you see a **ConvertTritonGPUToLLVM** error in the pod logs when querying the **/v1/chat/completions** API, and the vLLM server restarts or returns a **500 Internal Server** error, apply the following workaround:

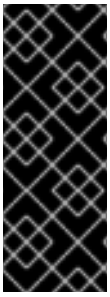
Before deploying the model, remove the **--enable-chunked-prefill** argument from the **Additional serving runtime arguments** field in the deployment dialog.

The error is displayed similar to the following:

```
/opt/vllm/lib64/python3.12/site-packages/vllm/attention/ops/prefix_prefill.py:36:0: error:
Failures have been detected while processing an MLIR pass pipeline
/opt/vllm/lib64/python3.12/site-packages/vllm/attention/ops/prefix_prefill.py:36:0: note:
Pipeline failed while executing [`ConvertTritonGPUToLLVM` on 'builtin.module' operation]:
reproducer generated at `std::errs, please share the reproducer above with Triton project.`
INFO:     10.129.2.8:0 - "POST /v1/chat/completions HTTP/1.1" 500 Internal Server Error
```

3.4. TESTING YOUR VLLM MODEL ENDPOINTS

To verify that your deployed Llama 3.2 model is accessible externally, ensure that your vLLM model server is exposed as a network endpoint. You can then test access to the model from outside both the OpenShift cluster and the OpenShift AI interface.



IMPORTANT

If you selected **Make deployed models available through an external route** during deployment, your vLLM model endpoint is already accessible outside the cluster. You do not need to manually expose the model server. Manually exposing vLLM model endpoints, for example, by using **oc expose**, creates an unsecured route unless you configure authentication. Avoid exposing endpoints without security controls to prevent unauthorized access.

Prerequisites

- You have cluster administrator privileges for your OpenShift cluster.
- You have logged in to Red Hat OpenShift AI.
- You have activated the Llama Stack Operator in OpenShift AI.
- You have deployed an inference model, for example, the llama-3.2-3b-instruct model.
- You have installed the OpenShift CLI (**oc**) as described in the appropriate documentation for your cluster:
  - Installing the OpenShift CLI for OpenShift Container Platform
  - Installing the OpenShift CLI for Red Hat OpenShift Service on AWS

Procedure

1. Open a new terminal window.
   a. Log in to your OpenShift cluster from the CLI:
   b. In the upper-right corner of the OpenShift web console, click your user name and select **Copy login command**.
   c. After you have logged in, click **Display token**.

d. Copy the **Log in with this token** command and paste it in the OpenShift CLI (**oc**).

```
$ oc login --token=<token> --server=<openshift_cluster_url>
```

2. If you enabled **Require token authentication** during model deployment, retrieve your token:

```
$ export MODEL_TOKEN=$(oc get secret default-name-llama-32-3b-instruct-sa -n <project name> --template={{ .data.token }} | base64 -d)
```

3. Obtain your model endpoint URL:

- If you enabled **Make deployed models available through an external route** during model deployment, click **Endpoint details** on the **Deployments** page in the OpenShift AI dashboard to obtain your model endpoint URL.
- In addition, if you did not enable **Require token authentication** during model deployment, you can also enter the following command to retrieve the endpoint URL:

```
$ export MODEL_ENDPOINT="https://$(oc get route llama-32-3b-instruct -n <project name> --template={{ .spec.host }})"
```

4. Test the endpoint with a sample chat completion request:

- If you did not enable **Require token authentication** during model deployment, enter a chat completion request. For example:

```
$ curl -X POST $MODEL_ENDPOINT/v1/chat/completions \
 -H "Content-Type: application/json" \
 -d '{
 "model": "llama-32-3b-instruct",
 "messages": [
  {
   "role": "user",
   "content": "Hello"
  }
 ]
}'
```

- If you enabled **Require token authentication** during model deployment, include a token in your request. For example:

```
curl -s -k $MODEL_ENDPOINT/v1/chat/completions \
--header "Authorization: Bearer $MODEL_TOKEN" \
--header 'Content-Type: application/json' \
-d '{
  "model": "llama-32-3b-instruct",
  "messages": [
   {
    "role": "user",
    "content": "can you tell me a funny joke?"
   }
  ]
}' | jq .
```

NOTE

The **-k** flag disables SSL verification and should only be used in test environments or with self-signed certificates.

Verification

Confirm that you received a JSON response containing a chat completion. For example:


```
{
  "id": "chatcmpl-05d24b91b08a4b78b0e084d4cc91dd7e",
  "object": "chat.completion",
  "created": 1747279170,
  "model": "llama-32-3b-instruct",
  "choices": [{
    "index": 0,
    "message": {
      "role": "assistant",
      "reasoning_content": null,
      "content": "Hello! It's nice to meet you. Is there something I can help you with or would you like to chat?",
      "tool_calls": []
    },
    "logprobs": null,
    "finish_reason": "stop",
    "stop_reason": null
  }],
  "usage": {
    "prompt_tokens": 37,
    "total_tokens": 62,
    "completion_tokens": 25,
    "prompt_tokens_details": null
  },
  "prompt_logprobs": null
}
```


If you do not receive a response similar to the example, verify that the endpoint URL and token are correct, and ensure your model deployment is running.

3.5. DEPLOYING A REMOTE MILVUS VECTOR DATABASE

To use Milvus as a remote vector database provider for Llama Stack in OpenShift AI, you must deploy Milvus and its required etcd service in your OpenShift project. This procedure shows how to deploy Milvus in standalone mode without the Milvus Operator.

NOTE

The following example configuration is intended for testing or evaluation environments. For production-grade deployments, see https://milvus.io/docs in the Milvus documentation.

Prerequisites

- You have installed OpenShift 4.19 or newer.

- You have enabled GPU support in OpenShift AI. This includes installing the Node Feature Discovery operator and NVIDIA GPU Operators. For more information, see Installing the Node Feature Discovery operator and Enabling NVIDIA GPUs.
- You have cluster administrator privileges for your OpenShift cluster.
- You are logged in to Red Hat OpenShift AI.
- You have a StorageClass available that can provision persistent volumes.
- You created a root password to secure your Milvus service.
- You have deployed an inference model with vLLM, for example, the llama-3.2-3b-instruct model, and you have selected **Make deployed models available through an external route** and **Require token authentication** during model deployment.
- You have the correct inference model identifier, for example, llama-3-2-3b.
- You have the model endpoint URL, ending with **/v1**, such as **https://llama-32-3b-instruct-predictor:8443/v1**.
- You have the API token required to access the model endpoint.
- You have installed the OpenShift command line interface (**oc**) as described in Installing the OpenShift CLI.

Procedure

1. In the OpenShift console, click the **Quick Create** () icon and then click the **Import YAML** option.
2. Verify that your project is the selected project.
3. In the **Import YAML** editor, paste the following manifest and click **Create**:

```
apiVersion: v1
kind: Secret
metadata:
  name: milvus-secret
type: Opaque
stringData:
  root-password: "MyStr0ngP@ssw0rd"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: milvus-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  volumeMode: Filesystem
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: etcd-deployment
  labels:
    app: etcd
spec:
  replicas: 1
  selector:
    matchLabels:
      app: etcd
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: etcd
    spec:
      containers:
        - name: etcd
          image: quay.io/coreos/etcd:v3.5.5
          command:
            - etcd
            - --advertise-client-urls=http://127.0.0.1:2379
            - --listen-client-urls=http://0.0.0.0:2379
            - --data-dir=/etcd
          ports:
            - containerPort: 2379
          volumeMounts:
            - name: etcd-data
              mountPath: /etcd
          env:
            - name: ETCD_AUTO_COMPACTION_MODE
              value: revision
            - name: ETCD_AUTO_COMPACTION_RETENTION
              value: "1000"
            - name: ETCD_QUOTA_BACKEND_BYTES
              value: "4294967296"
            - name: ETCD_SNAPSHOT_COUNT
              value: "50000"
      volumes:
        - name: etcd-data
          emptyDir: {}
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: etcd-service
spec:
  ports:
    - port: 2379
      targetPort: 2379
  selector:
    app: etcd
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: milvus-standalone
  name: milvus-standalone
spec:
  replicas: 1
  selector:
    matchLabels:
      app: milvus-standalone
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: milvus-standalone
    spec:
      containers:
        - name: milvus-standalone
          image: milvusdb/milvus:v2.6.0
          args: ["milvus", "run", "standalone"]
          env:
            - name: DEPLOY_MODE
              value: standalone
            - name: ETCD_ENDPOINTS
              value: etcd-service:2379
            - name: COMMON_STORAGETYPE
              value: local
            - name: MILVUS_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: milvus-secret
                  key: root-password
          livenessProbe:
            exec:
              command: ["curl", "-f", "http://localhost:9091/healthz"]
            initialDelaySeconds: 90
            periodSeconds: 30
            timeoutSeconds: 20
            failureThreshold: 5
          ports:
            - containerPort: 19530
              protocol: TCP
            - containerPort: 9091
              protocol: TCP
          volumeMounts:
            - name: milvus-data
              mountPath: /var/lib/milvus
      restartPolicy: Always
      volumes:
        - name: milvus-data
          persistentVolumeClaim:
            claimName: milvus-pvc
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: milvus-service
spec:
  selector:
    app: milvus-standalone
  ports:
    - name: grpc
      port: 19530
      targetPort: 19530
    - name: http
      port: 9091
      targetPort: 9091
```

NOTE

- Use the gRPC port (**19530**) for the **MILVUS_ENDPOINT** setting in Llama Stack.
- The HTTP port (**9091**) is reserved for health checks.
- If you deploy Milvus in a different namespace, use the fully qualified service name in your Llama Stack configuration. For example: **http://milvus-service.<namespace>.svc.cluster.local:19530**

Verification

1. In the OpenShift web console, click **Workloads → Deployments**.
2. Verify that both **etcd-deployment** and **milvus-standalone** show a status of **1 of 1 pods available**.
3. Click **Pods** in the navigation panel and confirm that pods for both deployments are **Running**.
4. Click the **milvus-standalone** pod name, then select the **Logs** tab.
5. Verify that Milvus reports a healthy startup with output similar to:

```
Milvus Standalone is ready to serve ...
Listening on 0.0.0.0:19530 (gRPC)
```

6. Click **Networking → Services** and confirm that the **milvus-service** and **etcd-service** resources exist and are exposed on ports **19530** and **2379**, respectively.
7. (Optional) Click **Pods → milvus-standalone → Terminal** and run the following health check:

```
curl http://localhost:9091/healthz
```

A response of **{"status": "healthy"}** confirms that Milvus is running correctly.

3.6. DEPLOYING A LLAMASTACKDISTRIBUTION INSTANCE

You can deploy Llama Stack with retrieval-augmented generation (RAG) by pairing it with a vLLM-served Llama 3.2 model. This module provides the following deployment examples of the **LlamaStackDistribution** custom resource (CR):

- **Example A:** Inline Milvus (embedded, single-node)
- **Example B:** Remote Milvus (external service)
- **Example C:** Inline FAISS (SQLite backend)

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have enabled GPU support in OpenShift AI. This includes installing the Node Feature Discovery Operator and NVIDIA GPU Operator. For more information, see Installing the Node Feature Discovery Operator and Enabling NVIDIA GPUs.
- You have cluster administrator privileges for your OpenShift cluster.
- You are logged in to Red Hat OpenShift AI.
- You have activated the Llama Stack Operator in OpenShift AI.
- You have deployed an inference model with vLLM (for example, **llama-3.2-3b-instruct**) and selected **Make deployed models available through an external route** and **Require token authentication** during model deployment. In addition, in **Add custom runtime arguments**, you have added **--enable-auto-tool-choice**.
- You have the correct inference model identifier, for example, **llama-3-2-3b**.
- You have the model endpoint URL ending with **/v1**, for example, **https://llama-32-3b-instruct-predictor:8443/v1**.
- You have the API token required to access the model endpoint.
- You have installed the OpenShift CLI (**oc**) as described in the appropriate documentation for your cluster:
  - Installing the OpenShift CLI for OpenShift Container Platform
  - Installing the OpenShift CLI for Red Hat OpenShift Service on AWS

Procedure

1. Open a new terminal window and log in to your OpenShift cluster from the CLI:
   In the upper-right corner of the OpenShift web console, click your user name and select **Copy login command**. After you have logged in, click **Display token**. Copy the **Log in with this token** command and paste it in the OpenShift CLI (**oc**).

   ```
   $ oc login --token=<token> --server=<openshift_cluster_url>
   ```

2. Create a secret that contains the inference model environment variables:

   ```
   export INFERENCE_MODEL="llama-3-2-3b"
   export VLLM_URL="https://llama-32-3b-instruct-predictor:8443/v1"
   ```

```
export VLLM_TLS_VERIFY="false"   # Use "true" in production
export VLLM_API_TOKEN="<token identifier>"

oc create secret generic llama-stack-inference-model-secret \
  --from-literal=INFERENCE_MODEL="$INFERENCE_MODEL" \
  --from-literal=VLLM_URL="$VLLM_URL" \
  --from-literal=VLLM_TLS_VERIFY="$VLLM_TLS_VERIFY" \
  --from-literal=VLLM_API_TOKEN="$VLLM_API_TOKEN"
```

3. Choose **one** of the following deployment examples:

IMPORTANT

To enable Llama Stack in a disconnected environment, you need add the following parameters to your **LlamaStackDistribution** custom resource.

```
- name: SENTENCE_TRANSFORMERS_HOME
  value: /opt/app-root/src/.cache/huggingface/hub
- name: HF_HUB_OFFLINE
  value: "1"
- name: TRANSFORMERS_OFFLINE
  value: "1"
- name: HF_DATASETS_OFFLINE
  value: "1"
```

The built-in Llama Stack tool **websearch** is not available in the Red Hat Llama Stack Distribution in disconnected environments. In addition, the built-in Llama Stack tool **wolfram_alpha** tool is not available in the Red Hat Llama Stack Distribution in all clusters.

3.6.1. Example A: LlamaStackDistribution with Inline Milvus

Use this example for development or small datasets where an embedded, single-node Milvus is sufficient. No **MILVUS_*** connection variables are required.

1. In the OpenShift web console, select **Administrator → Quick Create** () **→ Import YAML**, and create a CR similar to the following:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: lsd-llama-milvus-inline
spec:
  replicas: 1
  server:
    containerSpec:
      resources:
        requests:
          cpu: "250m"
          memory: "500Mi"
        limits:
          cpu: 4
          memory: "12Gi"
      env:
```

```
      - name: INFERENCE_MODEL
        valueFrom:
          secretKeyRef:
            name: llama-stack-inference-model-secret
            key: INFERENCE_MODEL
      - name: VLLM_MAX_TOKENS
        value: "4096"
      - name: VLLM_URL
        valueFrom:
          secretKeyRef:
            name: llama-stack-inference-model-secret
            key: VLLM_URL
      - name: VLLM_TLS_VERIFY
        valueFrom:
          secretKeyRef:
            name: llama-stack-inference-model-secret
            key: VLLM_TLS_VERIFY
      - name: VLLM_API_TOKEN
        valueFrom:
          secretKeyRef:
            name: llama-stack-inference-model-secret
            key: VLLM_API_TOKEN
    name: llama-stack
    port: 8321
  distribution:
    name: rh-dev
```

NOTE

The **rh-dev** value is an internal image reference. When you create the **LlamaStackDistribution** custom resource, the OpenShift AI Operator automatically resolves **rh-dev** to the container image in the appropriate registry. This internal image reference allows the underlying image to update without requiring changes to your custom resource.

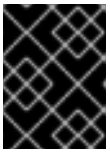
3.6.2. Example B: LlamaStackDistribution with Remote Milvus

Use this example for production-grade or large datasets with an external Milvus service. This configuration reads both **MILVUS_ENDPOINT** and **MILVUS_TOKEN** from a dedicated secret.

1. Create the Milvus connection secret:

```
# Required: gRPC endpoint on port 19530
export MILVUS_ENDPOINT="tcp://milvus-service:19530"
export MILVUS_TOKEN="<milvus-root-or-user-token>"
export MILVUS_CONSISTENCY_LEVEL="Bounded"   # Optional; choose per your deployment

oc create secret generic milvus-secret \
  --from-literal=MILVUS_ENDPOINT="$MILVUS_ENDPOINT" \
  --from-literal=MILVUS_TOKEN="$MILVUS_TOKEN" \
  --from-literal=MILVUS_CONSISTENCY_LEVEL="$MILVUS_CONSISTENCY_LEVEL"
```



IMPORTANT

Use the **gRPC port19530** for **MILVUS_ENDPOINT**. Ports such as **9091** are typically used for health checks and are not valid for client traffic.

2. In the OpenShift web console, select **Administrator → Quick Create** () **→ Import YAML**, and create a CR similar to the following:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: lsd-llama-milvus-remote
spec:
  replicas: 1
  server:
    containerSpec:
      resources:
        requests:
          cpu: "250m"
          memory: "500Mi"
        limits:
          cpu: 4
          memory: "12Gi"
      env:
        - name: INFERENCE_MODEL
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: INFERENCE_MODEL
        - name: VLLM_MAX_TOKENS
          value: "4096"
        - name: VLLM_URL
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: VLLM_URL
        - name: VLLM_TLS_VERIFY
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: VLLM_TLS_VERIFY
        - name: VLLM_API_TOKEN
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: VLLM_API_TOKEN
        # --- Remote Milvus configuration from secret ---
        - name: MILVUS_ENDPOINT
          valueFrom:
            secretKeyRef:
              name: milvus-secret
              key: MILVUS_ENDPOINT
        - name: MILVUS_TOKEN
          valueFrom:
            secretKeyRef:
```

```
            name: milvus-secret
            key: MILVUS_TOKEN
        - name: MILVUS_CONSISTENCY_LEVEL
          valueFrom:
            secretKeyRef:
              name: milvus-secret
              key: MILVUS_CONSISTENCY_LEVEL
      name: llama-stack
      port: 8321
    distribution:
      name: rh-dev
```

3.6.3. Example C: LlamaStackDistribution with Inline FAISS

Use this example to enable the inline FAISS vector store. This configuration stores vector data locally within the Llama Stack container using an embedded SQLite database.

1. In the OpenShift web console, select **Administrator → Quick Create** () **→ Import YAML**, and create a CR similar to the following:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: lsd-llama-faiss-inline
spec:
  replicas: 1
  server:
    containerSpec:
      resources:
        requests:
          cpu: "250m"
          memory: "500Mi"
        limits:
          cpu: "8"
          memory: "12Gi"
      env:
        # vLLM inference model configuration
        - name: INFERENCE_MODEL
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: INFERENCE_MODEL
        - name: VLLM_URL
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: VLLM_URL
        - name: VLLM_TLS_VERIFY
          valueFrom:
            secretKeyRef:
              name: llama-stack-inference-model-secret
              key: VLLM_TLS_VERIFY
        - name: VLLM_API_TOKEN
          valueFrom:
```

```
      secretKeyRef:
        name: llama-stack-inference-model-secret
        key: VLLM_API_TOKEN

    # Enable inline FAISS with SQLite backend
    - name: ENABLE_FAISS
      value: faiss
    - name: FAISS_KVSTORE_DB_PATH
      value: /opt/app-root/src/.llama/distributions/rh/sqlite_vec.db

    # Recommended workarounds for SQLite accessibility and version check
    - name: LLAMA_STACK_CONFIG_DIR
      value: /opt/app-root/src/.llama/distributions/rh
  name: llama-stack
  port: 8321
distribution:
  name: rh-dev
```

NOTE

The **FAISS_KVSTORE_DB_PATH** environment variable defines the local path where the FAISS SQLite backend stores its index data. Ensure that this directory exists and is writable inside the container. Inline FAISS is only suitable for experimental or testing use cases.

1. Click **Create**.

Verification

- In the left-hand navigation, click **Workloads → Pods** and verify that the Llama Stack pod is running in the correct namespace.
- To verify that the Llama Stack server is running, click the pod name and select the **Logs** tab. Look for output similar to the following:

```
INFO     2025-05-15 11:23:52,750 __main__:498 server: Listening on ['::', '0.0.0.0']:8321
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO     2025-05-15 11:23:52,765 __main__:151 server: Starting up
INFO:     Application startup complete.
INFO:     Uvicorn running on http://['::', '0.0.0.0']:8321 (Press CTRL+C to quit)
```

- Confirm that a Service resource for the Llama Stack backend is present in your namespace and points to the running pod: **Networking → Services**.

TIP

If you switch from Inline Milvus to Remote Milvus, delete the existing pod to ensure the new environment variables and backing store are picked up cleanly.

3.7. INGESTING CONTENT INTO A LLAMA MODEL

You can quickly customize and prototype your retrievable content by ingesting raw text into your model from inside a Jupyter notebook. This approach avoids building a separate ingestion pipeline. By using

the Llama Stack SDK, you can embed and store text in your vector store in real time, enabling immediate RAG workflows.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have deployed a Llama 3.2 model with a vLLM model server.
- You have created a **LlamaStackDistribution** instance (Llama Stack).
- You have created a workbench within a project.
- You have opened a Jupyter notebook and it is running in your workbench environment.
- You have installed the **llama_stack_client** version 0.3.1 or later in your workbench environment.
- If you use a remote vector store, your environment has network access to that service through OpenShift.

Procedure

1. In a new notebook cell, install the client:

```
%pip install llama_stack_client
```

2. Import **LlamaStackClient** and create a client instance:

```
from llama_stack_client import LlamaStackClient
client = LlamaStackClient(base_url="<your deployment endpoint>")
```

3. List the available models:

```
# Fetch all registered models
models = client.models.list()
```

4. Verify that the list includes your Llama model and an embedding model. For example:

```
[Model(identifier='llama-32-3b-instruct', metadata={}, api_model_type='llm', provider_id='vllm-inference', provider_resource_id='llama-32-3b-instruct', type='model', model_type='llm'),
 Model(identifier='ibm-granite/granite-embedding-125m-english', metadata={'embedding_dimension': 768.0}, api_model_type='embedding', provider_id='sentence-transformers', provider_resource_id='ibm-granite/granite-embedding-125m-english', type='model', model_type='embedding')]
```

5. Select one LLM and one embedding model:

```
model_id = next(m.identifier for m in models if m.model_type == "llm")
embedding_model = next(m for m in models if m.model_type == "embedding")
embedding_model_id = embedding_model.identifier
embedding_dimension = int(embedding_model.metadata["embedding_dimension"])
```

6. (Optional) Create a vector store (choose one). Skip this step if you already have one.

Example 3.1. Option 1: Inline Milvus Lite (embedded)

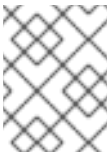
```
vector_store_name = "my_inline_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus",   # inline Milvus Lite
    },
)
vector_store_id = vector_store.id
print(f"Registered inline Milvus Lite DB: {vector_store_id}")
```

NOTE

Use inline Milvus Lite for development and small datasets. Persistence and scale are limited compared to remote Milvus.

Example 3.2. Option 2: Remote Milvus (recommended for production)

```
vector_store_name = "my_remote_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus-remote",  # remote Milvus provider
    },
)
vector_store_id = vector_store.id
print(f"Registered remote Milvus DB: {vector_store_id}")
```



NOTE

Ensure your **LlamaStackDistribution** sets **MILVUS_ENDPOINT** (gRPC **:19530**) and **MILVUS_TOKEN**.

Example 3.3. Option 3: Inline FAISS (SQLite backend)

```
vector_store_name = "my_faiss_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "faiss",   # inline FAISS provider
    },
```

```
)
vector_store_id = vector_store.id
print(f"Registered inline FAISS DB: {vector_store_id}")
```

NOTE

Inline FAISS (available in OpenShift AI 3.0 and later) is a lightweight, in-process vector store with SQLite-based persistence. It is best for local experimentation, disconnected environments, or single-node RAG deployments.

1. If you already have a vector store, set its identifier:

```
# For an existing vector store:
# vector_store_id = "<your existing vector store ID>"
```

2. Define raw text to ingest:

```
raw_text = """Llama Stack can embed raw text into a vector store for retrieval.
This example ingests a small passage for demonstration."""
```

3. Ingest raw text by using the Vector Store Files API:

```
items = [
    {
        "id": "raw_text_001",
        "text": raw_text,
        "mime_type": "text/plain",
        "metadata": {"source": "example_passage"},
    }
]
result = client.vector_stores.files.create(
    vector_store_id=vector_store_id,
    items=items,
    chunk_size_in_tokens=100,
)
print("Text ingestion result:", result)
```

4. Ingest an HTML source:

```
html_item = [
    {
        "id": "doc_html_001",
        "text": "https://www.paulgraham.com/greatwork.html",
        "mime_type": "text/html",
        "metadata": {"note": "Example URL"},
    }
]
result = client.vector_stores.files.create(
    vector_store_id=vector_store_id,
    items=html_item,
    chunk_size_in_tokens=50,
)
print("HTML ingestion result:", result)
```

Verification

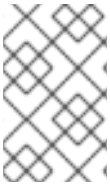
- Review the output to confirm successful ingestion. A typical response includes file or chunk counts and any warnings or errors.
- The model list returned by **client.models.list()** includes your Llama 3.2 model and an embedding model.

3.8. QUERYING INGESTED CONTENT IN A LLAMA MODEL

You can use the Llama Stack SDK in your Jupyter notebook to query ingested content by running retrieval-augmented generation (RAG) queries on text or HTML stored in your vector store. You can perform one-off lookups or start multi-turn conversational flows without setting up a separate retrieval service.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have enabled GPU support in OpenShift AI. This includes installing the Node Feature Discovery operator and NVIDIA GPU Operators. For more information, see Installing the Node Feature Discovery operator and Enabling NVIDIA GPUs.
- If you are using GPU acceleration, you have at least one NVIDIA GPU available.
- You have activated the Llama Stack Operator in OpenShift AI.
- You have deployed an inference model, for example, the **llama-3.2-3b-instruct** model.
- You have created a **LlamaStackDistribution** instance to enable RAG functionality.
- You have created a workbench within a project and opened a running Jupyter notebook.
- You have installed **llama_stack_client** version 0.3.1 or later in your workbench environment.
- You have already ingested content into a vector store.



NOTE

This procedure requires that you have already ingested some text, HTML, or document data into a vector store, and that this content is available for retrieval. If no content is ingested, queries return empty results.

Procedure

1. In a new notebook cell, install the client:

```
%pip install -q llama_stack_client
```

2. In a new notebook cell, import **Agent**, **AgentEventLogger**, and **LlamaStackClient**:

```
from llama_stack_client import Agent, AgentEventLogger, LlamaStackClient
```

3. Create a client instance by setting your deployment endpoint:

```
client = LlamaStackClient(base_url="<your deployment endpoint>")
```

4. List available models:

```
models = client.models.list()
```

5. Select an LLM (and, if needed below, capture an embedding model for store registration):

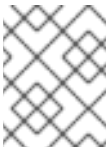
```
model_id = next(m.identifier for m in models if m.model_type == "llm")

embedding = next((m for m in models if m.model_type == "embedding"), None)
if embedding:
    embedding_model_id = embedding.identifier
    embedding_dimension = int(embedding.metadata.get("embedding_dimension", 768))
```

6. If you do not already have a vector store ID, register a vector store (choose one):

Example 3.4. Option 1: Inline Milvus Lite (embedded)

```
vector_store_name = "my_inline_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus",   # inline Milvus Lite
    },
)
vector_store_id = vector_store.id
print(f"Registered inline Milvus Lite DB: {vector_store_id}")
```

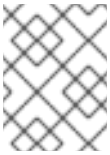


NOTE

Use inline Milvus Lite for development and small datasets. Persistence and scale are limited compared to remote Milvus.

Example 3.5. Option 2: Remote Milvus (recommended for production)

```
vector_store_name = "my_remote_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus-remote",  # remote Milvus provider
    },
)
vector_store_id = vector_store.id
print(f"Registered remote Milvus DB: {vector_store_id}")
```

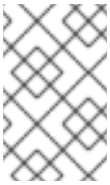


NOTE

Ensure your **LlamaStackDistribution** sets **MILVUS_ENDPOINT** (gRPC **:19530**) and **MILVUS_TOKEN**.

Example 3.6. Option 3: Inline FAISS (SQLite backend)

```
vector_store_name = "my_faiss_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "faiss",   # inline FAISS provider
    },
)
vector_store_id = vector_store.id
print(f"Registered inline FAISS DB: {vector_store_id}")
```



NOTE

Inline FAISS (available in OpenShift AI 3.0 and later) is a lightweight, in-process vector store with SQLite-based persistence. It is best for local experimentation, disconnected environments, or single-node RAG deployments.

1. If you already have a vector store, set its identifier:

```
# For an existing store:
# vector_store_id = "<your existing vector store ID>"
```

2. Query the ingested content by using the OpenAI-compatible Responses API with file search:

```
query = "What benefits do the ingested passages provide for retrieval?"

response = client.responses.create(
    model=model_id,
    input=query,
    tools=[
        {
            "type": "file_search",
            "vector_store_ids": [vector_store_id],
        }
    ],
)
print("Responses API result:", getattr(response, "output_text", response))
```

3. Query the ingested content by using the high-level Agent API:

```
agent = Agent(
    client,
    model=model_id,
```

```
    instructions="You are a helpful assistant.",
    tools=[
        {
            "name": "builtin::rag/knowledge_search",
            "args": {"vector_store_ids": [vector_store_id]},
        }
    ],
)

prompt = "How do you do great work?"
print("Prompt>", prompt)

session_id = agent.create_session("rag_session")
stream = agent.create_turn(
    messages=[{"role": "user", "content": prompt}],
    session_id=session_id,
    stream=True,
)

for log in AgentEventLogger().log(stream):
    log.print()
```

Verification

- The notebook prints query results for both the Responses API and the Agent API.
- No errors appear in the output, confirming the model can retrieve and respond to ingested content from your vector store.

3.9. PREPARING DOCUMENTS WITH DOCLING FOR LLAMA STACK RETRIEVAL

You can transform your source documents with a Docling-enabled pipeline and ingest the output into a Llama Stack vector store by using the Llama Stack SDK. This modular approach separates document preparation from ingestion, yet still delivers an end-to-end, retrieval-augmented generation (RAG) workflow.

The pipeline registers a vector store and downloads the source PDFs, then splits them for parallel processing and converts each batch to Markdown with Docling. It generates sentence-transformer embeddings from the Markdown and stores them in the vector store, making the documents searchable through Llama Stack.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have enabled GPU support in OpenShift AI. This includes installing the Node Feature Discovery operator and NVIDIA GPU Operators. For more information, see Installing the Node Feature Discovery operator and Enabling NVIDIA GPUs.
- You have logged in to the OpenShift web console.
- You have a project and access to pipelines in the OpenShift AI dashboard.

- You have created and configured a pipeline server within the project that contains your workbench.
- You have activated the Llama Stack Operator in OpenShift AI.
- You have deployed an inference model, for example, the **llama-3.2-3b-instruct** model.
- You have configured a Llama Stack deployment by creating a **LlamaStackDistribution** instance to enable RAG functionality.
- You have created a workbench within a project.
- You have opened a Jupyter notebook and it is running in your workbench environment.
- You have installed the **llama_stack_client** version 0.3.1 or later in your workbench environment.
- You have installed local object storage buckets and created connections, as described in Adding a connection to your project.
- You have compiled to YAML a pipeline that includes a Docling transform, either one of the RAG demo samples or your own custom pipeline.
- Your project quota allows between 500 millicores (0.5 CPU) and 4 CPU cores for the pipeline run.
- Your project quota allows from 2 GiB up to 6 GiB of RAM for the pipeline run.
- If you are using GPU acceleration, you have at least one NVIDIA GPU available.

Procedure

1. In a new notebook cell, install the client:

```
%pip install -q llama_stack_client
```

2. In a new notebook cell, import **Agent**, **AgentEventLogger**, and **LlamaStackClient**:

```
from llama_stack_client import Agent, AgentEventLogger, LlamaStackClient
```

3. In a new notebook cell, assign your deployment endpoint to the **base_url** parameter to create a **LlamaStackClient** instance:

```
client = LlamaStackClient(base_url="<your deployment endpoint>")
```

4. List the available models:

```
models = client.models.list()
```

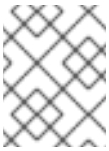
5. Select the first LLM and the first embedding model:

```
model_id = next(m.identifier for m in models if m.model_type == "llm")
embedding_model = next(m for m in models if m.model_type == "embedding")
embedding_model_id = embedding_model.identifier
embedding_dimension = int(embedding_model.metadata.get("embedding_dimension", 768))
```

6. Register a vector store (choose one option). Skip this step if your pipeline registers the store automatically.

Example 3.7. Option 1: Inline Milvus Lite (embedded)

```
vector_store_name = "my_inline_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus",   # inline Milvus Lite
    },
)
vector_store_id = vector_store.id
print(f"Registered inline Milvus Lite DB: {vector_store_id}")
```

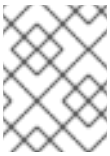


NOTE

Inline Milvus Lite is best for development. Data durability and scale are limited compared to remote Milvus.

Example 3.8. Option 2: Remote Milvus (recommended for production)

```
vector_store_name = "my_remote_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "milvus-remote",  # remote Milvus provider
    },
)
vector_store_id = vector_store.id
print(f"Registered remote Milvus DB: {vector_store_id}")
```



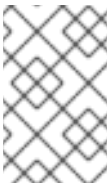
NOTE

Ensure your **LlamaStackDistribution** includes **MILVUS_ENDPOINT** and **MILVUS_TOKEN** (gRPC **:19530**).

Example 3.9. Option 3: Inline FAISS (SQLite backend)

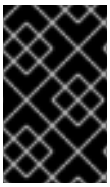
```
vector_store_name = "my_faiss_db"
vector_store = client.vector_stores.create(
    name=vector_store_name,
    extra_body={
        "embedding_model": embedding_model_id,
        "embedding_dimension": embedding_dimension,
        "provider_id": "faiss",   # inline FAISS provider
    },
```

```
)
vector_store_id = vector_store.id
print(f"Registered inline FAISS DB: {vector_store_id}")
```



NOTE

Inline FAISS (available in OpenShift AI 3.0 and later) is a lightweight, in-process vector store with SQLite-based persistence. It is best for local experimentation, disconnected environments, or single-node RAG deployments.



IMPORTANT

If you are using the sample Docling pipeline from the RAG demo repository, the pipeline registers the vector store automatically and you can skip the previous step. If you are using your own pipeline, you must register the vector store yourself.

1. In the OpenShift web console, import the YAML file containing your Docling pipeline into your project, as described in Importing a pipeline.
2. Create a pipeline run to execute your Docling pipeline, as described in Executing a pipeline run. The pipeline run inserts your PDF documents into the vector store. If you run the Docling pipeline from the RAG demo samples repository, you can optionally customize the following parameters before starting the pipeline run:
   - **base_url**: The base URL to fetch PDF files from.
   - **pdf_filenames**: A comma-separated list of PDF filenames to download and convert.
   - **num_workers**: The number of parallel workers.
   - **vector_store_id**: The vector store identifier.
   - **service_url**: The Milvus service URL (only for remote Milvus).
   - **embed_model_id**: The embedding model to use.
   - **max_tokens**: The maximum tokens for each chunk.
   - **use_gpu**: Enable or disable GPU acceleration.

Verification

1. In your Jupyter notebook, query the LLM with a question that relates to the ingested content. For example:

```
from llama_stack_client import Agent, AgentEventLogger
import uuid

rag_agent = Agent(
    client,
    model=model_id,
    instructions="You are a helpful assistant",
    tools=[
```

```
        {
            "name": "builtin::rag/knowledge_search",
            "args": {"vector_store_ids": [vector_store_id]},
        }
    ],
)

prompt = "What can you tell me about the birth of word processing?"
print("prompt>", prompt)

session_id = rag_agent.create_session(session_name=f"s{uuid.uuid4().hex}")

response = rag_agent.create_turn(
    messages=[{"role": "user", "content": prompt}],
    session_id=session_id,
    stream=True,
)

for log in AgentEventLogger().log(response):
    log.print()
```

2. Query chunks from the vector store:

```
query_result = client.vector_io.query(
    vector_store_id=vector_store_id,
    query="what do you know about?",
)
print(query_result)
```

Verification

- The pipeline run completes successfully in your project.
- Document embeddings are stored in the vector store and are available for retrieval.
- No errors or warnings appear in the pipeline logs or your notebook output.

3.10. ABOUT LLAMA STACK SEARCH TYPES

Llama Stack supports keyword, vector, and hybrid search modes for retrieving context in retrieval-augmented generation (RAG) workloads. Each mode offers different tradeoffs in precision, recall, semantic depth, and computational cost.

3.10.1. Supported search modes

3.10.1.1. Keyword search

Keyword search applies lexical matching techniques, such as TF-IDF or BM25, to locate documents that contain exact or near-exact query terms. This approach is effective when precise term-matching is critical and remains widely used in information-retrieval systems. For more information, see The Probabilistic Relevance Framework: BM25 and Beyond.

3.10.1.2. Vector search

Vector search encodes documents and queries as dense numerical vectors, known as embeddings, and measures similarity with metrics such as cosine similarity or inner product. This approach captures contextual meaning and supports semantic matching beyond exact word overlap. For more information, see Billion-scale similarity search with GPUs.

3.10.1.3. Hybrid search

Hybrid search blends keyword and vector techniques, typically by combining individual scores with a weighted sum or methods, such as Reciprocal Rank Fusion (RRF). This approach returns results that balance exact matches with semantic relevance. For more information, see Sparse, Dense, and Hybrid Retrieval for Answer Ranking.

3.10.2. Retrieval database support

Milvus is the supported retrieval database for Llama Stack. It currently provides vector search. However, keyword and hybrid search capabilities are not currently supported.

CHAPTER 4. EVALUATING RAG SYSTEMS WITH LLAMA STACK

You can use the evaluation providers that Llama Stack exposes to measure and improve the quality of your Retrieval-Augmented Generation (RAG) workloads in OpenShift AI. This section introduces RAG evaluation providers, describes how to use Ragas with Llama Stack, shows how to benchmark embedding models with BEIR, and helps you choose the right provider for your use case.

4.1. UNDERSTANDING RAG EVALUATION PROVIDERS

Llama Stack supports pluggable evaluation providers that measure the quality and performance of Retrieval-Augmented Generation (RAG) pipelines. Evaluation providers assess how accurately, faithfully, and relevantly the generated responses align with the retrieved context and the original user query. Each provider implements its own metrics and evaluation methodology. You can enable a specific provider through the configuration of the **LlamaStackDistribution** custom resource.

OpenShift AI supports the following evaluation providers:

- **Ragas**: A lightweight, Python-based framework that evaluates factuality, contextual grounding, and response relevance.
- **BEIR**: A benchmarking framework for retrieval performance across multiple datasets.
- **TrustyAI**: A Red Hat framework that evaluates explainability, fairness, and reliability of model outputs.

Evaluation providers operate independently of model serving and retrieval components. You can run evaluations asynchronously and aggregate results for quality tracking over time.

4.2. USING RAGAS WITH LLAMA STACK

You can use the Ragas (Retrieval-Augmented Generation Assessment) evaluation provider with Llama Stack to measure the quality of your Retrieval-Augmented Generation (RAG) workflows in OpenShift AI. Ragas integrates with the Llama Stack evaluation API to compute metrics such as faithfulness, answer relevancy, and context precision for your RAG workloads.

Llama Stack exposes evaluation providers as part of its API surface. When you configure Ragas as a provider, the Llama Stack server sends RAG inputs and outputs to Ragas and records the resulting metrics for later analysis.

Ragas evaluation with Llama Stack in OpenShift AI supports the following deployment modes:

- **Inline provider** for development and small-scale experiments.
- **Remote provider** for production-scale evaluations that run as OpenShift AI AI pipelines.

You choose the mode that best fits your workflow:

- Use the inline provider when you want fast, low-overhead evaluation while you iterate on prompts, retrieval configuration, or model choices.
- Use the remote provider when you need to evaluate large datasets, integrate with CI/CD pipelines, or run repeated benchmarks at scale.

For information on evaluating RAG systems with Ragas in OpenShift AI, see Evaluating RAG systems with RAGAS

4.3. BENCHMARKING EMBEDDING MODELS WITH BEIR DATASETS AND LLAMA STACK

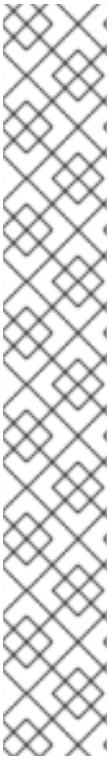
This procedure explains how to set up, run, and verify embedding-model benchmarks by using the Llama Stack framework. Embedding models are neural networks that convert text or other data into dense numerical vectors, called embeddings, which capture semantic meaning. In retrieval-augmented generation (RAG) systems, embeddings enable semantic search so that the system retrieves the documents most relevant to a query.

Selecting an embedding model depends on several factors, such as the content type, accuracy requirements, performance needs, and model license. The **beir_benchmarks.py** script compares the retrieval accuracy of embedding models by using standardized information-retrieval benchmarks from the BEIR framework. The script is included in the RAG repository, which provides demonstrations, benchmarking scripts, and deployment guides for the RAG Stack on OpenShift.

The examples use the **sentence-transformers** inference provider, which you can replace with another provider if required.

Prerequisites

- You have cloned the **https://github.com/opendatahub-io/rag** repository.
- You have changed into the **/rag/benchmarks/beir-benchmarks** directory.
- You have initialized and activated a virtual environment.
- You have defined and installed the relevant script package dependencies to a **requirements.txt** file.
- You have built the Llama Stack starter distribution to install all dependencies.
- You have verified that your vector database is accessible and configured in the **run.yaml** file, and that any required embedding models were preloaded or registered with Llama Stack.



NOTE

The default supported embedding models are **granite-embedding-30m** and **granite-embedding-125m**, served by the **sentence-transformers** framework. Ollama is not required for basic benchmarks but can be used to serve custom embedding models.

To register an additional embedding model, such as **all-MiniLM-L6-v2**, perform the following steps:

1. Start the Llama Stack server:

```
MILVUS_URL=milvus uv run llama stack run run.yaml
```

2. Register the model by using the Llama Stack client. For example:

```
llama-stack-client models register all-MiniLM-L6-v2 \
  --provider-id sentence-transformers \
  --provider-model-id all-minilm:latest \
  --metadata {"embedding_dimension": 384} \
  --model-type embedding
```

- You have shut down the Llama Stack server before running the benchmark script.

Procedure

1. Run the **beir_benchmarks.py** benchmarking script:
   - Enter the following command to use the configuration from **run.yaml** and the default dataset (**scifact**):

```
MILVUS_URL=milvus uv run python beir_benchmarks.py
```

   - Alternatively, enter the following command to connect to a custom Llama Stack server:

```
LLAMA_STACK_URL="http://localhost:8321" MILVUS_URL=milvus uv run python
beir_benchmarks.py
```

2. Use environment variables and command-line options to modify the benchmark run. For example, set the environment variable **ENABLE_MILVUS=milvus** before executing the script.
   - Enter the following command to benchmark with a specific LLM by using default settings:

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py
```

   - Enter the following command to use a larger batch size for document ingestion:

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py --batch-size 300
```

   - Enter the following command to benchmark multiple datasets (for example, **scifact** and **scidocs**):

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py \
  --dataset-names scifact scidocs
```

-

- Enter the following command to compare embedding models (for example, **granite-embedding-30m** and **all-MiniLM-L6-v2**):

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py \
  --embedding-models granite-embedding-30m all-MiniLM-L6-v2
```

- Enter the following command to use a custom BEIR-compatible dataset:

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py \
  --dataset-names my-dataset \
  --custom-datasets-urls https://example.com/my-beir-dataset.zip
```

- Enter the following command to change the vector database provider. The following example changes the vector database provider to remote Milvus:

```
ENABLE_MILVUS=milvus uv run python beir_benchmarks.py \
  --vector-db-provider-id remote-milvus
```

Command-line options

- **--vector-db-provider-id**
  - **Description:** Specifies the vector database provider to use.
  - **Type:** String.
  - **Default: milvus**.
  - **Example:**

```
--vector-db-provider-id remote-milvus
```

- **--dataset-names**
  - **Description:** Specifies which BEIR datasets to use for benchmarking. Use this option together with **--custom-datasets-urls** when testing custom datasets.
  - **Type:** List of strings.
  - **Default: ["scifact"]**.
  - **Example:**

```
--dataset-names scifact scidocs nq
```

- **--embedding-models**
  - **Description:** Specifies the embedding models to compare. Models must be defined in the **run.yaml** file.
  - **Type:** List of strings.
  - **Default: ["granite-embedding-30m", "granite-embedding-125m"]**.

- Example:

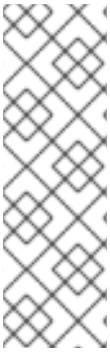
```
--embedding-models all-MiniLM-L6-v2 granite-embedding-125m
```

- **--batch-size**
  - **Description**: Controls how many documents are processed per batch during ingestion. Larger batch sizes improve speed but use more memory.
  - **Type:** Integer.
  - **Default: 150**.
  - **Example:**

```
--batch-size 50
--batch-size 300
```

- **--custom-datasets-urls**
  - **Description:** Specifies URLs for custom BEIR-compatible datasets. Use this option with **--dataset-names**.
  - **Type:** List of strings.
  - **Default: []**.
  - **Example:**

```
--dataset-names my-custom-dataset \
 --custom-datasets-urls https://example.com/my-dataset.zip
```



NOTE

Custom BEIR datasets must follow the required file structure and format:

```
dataset-name.zip/
 ├─── qrels/
 │    └─── test.tsv      # Maps query IDs to document IDs with relevance scores
 ├─── corpus.jsonl       # Document collection with document IDs, titles, and text
 └─── queries.jsonl      # Test queries with query IDs and question text
```

Verification

To verify that the benchmark completed successfully and to review the results, perform the following steps:

1. Locate the **results** directory. All output files are saved to the following path: **<path-to>/rag/benchmarks/embedding-models-with-beir/results**
2. Examine the output. Compare your results with the sample output structure. The report includes performance metrics such as **map@cut_k** and **ndcg@cut_k** for each dataset and embedding model pair. The script also calculates a statistical significance test (**p**-value). **Example output (for scifact and map_cut_10):**

```
scifact map_cut_10
 granite-embedding-125m : 0.6879
 granite-embedding-30m  : 0.6578
 p_value                : 0.0150

 p_value < 0.05 indicates a statistically significant difference.
 The granite-embedding-125m model performs better for this dataset and metric.
```

3. Interpret the results. A **p**-value below **0.05** indicates that the performance difference between models is statistically significant. Use these results to identify which embedding model performs best for your dataset.

CHAPTER 5. CONFIGURING LLAMA STACK WITH OAUTH AUTHENTICATION

You can configure Llama Stack to enable Role-Based Access Control (RBAC) for model access using OAuth authentication on OpenShift AI. The following example shows how to configure Llama Stack so that a vLLM model can be accessed by all authenticated users, while an OpenAI model is restricted to specific users. This example uses Keycloak to issue and validate tokens.

Before you begin, you must already have Keycloak set up with the following parameters:

- Realm: **llamastack-demo**.
- Client: **llamastack** with direct access grants enabled.
- Role: **inference_max** grants access to restricted models and a protocol mapper that adds realm roles to the access token under the claim name **llamastack_roles**.
- Two test users:
  - **user1** as a basic user with no assigned roles.
  - **user2** as an advanced user assigned the **inference_max** role.
- The client secret generated by Keycloak must be saved as you will need it for token requests.

This document assumes the Keycloak server is available at **https://my-keycloak-server.com**.

Prerequisites

- You have installed OpenShift 4.19 or newer.
- You have logged in to Red Hat OpenShift AI.
- You have cluster administrator privileges for your OpenShift cluster.
- You have installed the OpenShift CLI (**oc**) as described in the appropriate documentation for your cluster:
  - Installing the OpenShift CLI for OpenShift Container Platform
  - Installing the OpenShift CLI for Red Hat OpenShift Service on AWS

Procedure

1. To configure Llama Stack to use Role-Based Access Control (RBAC) to access models, you must view and verify the OAuth provider token structure.
   a. Generate a Keycloak test token to view the structure with the following command:

```
$ curl -d client_id=llamastack -d client_secret=YOUR_CLIENT_SECRET -d
username=user1 -d password=user-password -d grant_type=password https://my-
keycloak-server.com/realms/llamastack-demo/protocol/openid-connect/token | jq -r
.access_token > test.token
```

   b. View the token claims by running the following command:

```
$ cat test.token | cut -d . -f 2 | base64 -d 2>/dev/null | jq .
```

Example token structure from Keycloak

```
$ {
  "iss": "http://my-keycloak-server.com/realms/llamastack-demo",
  "aud": "account",
  "sub": "761cdc99-80e5-4506-9b9e-26a67a8566f7",
  "preferred_username": "user1",
  "llamastack_roles": [
    "inference_max"
  ],
}
```

2. Create a **run.yaml** file that defines the necessary configurations for OAuth.

   a. Define a configuration with two inference providers and OAuth authentication with the following **run.yaml** example:

```
version: 2
image_name: rh
apis:
  - inference
  - agents
  - safety
  - telemetry
  - tool_runtime
  - vector_io
providers:
  inference:
    - provider_id: vllm-inference
      provider_type: remote::vllm
      config:
        url: ${env.VLLM_URL:=http://localhost:8000/v1}
        max_tokens: ${env.VLLM_MAX_TOKENS:=4096}
        api_token: ${env.VLLM_API_TOKEN:=fake}
        tls_verify: ${env.VLLM_TLS_VERIFY:=true}
    - provider_id: openai
      provider_type: remote::openai
      config:
        api_key: ${env.OPENAI_API_KEY:=}
        base_url: ${env.OPENAI_BASE_URL:=https://api.openai.com/v1}
  telemetry:
  - provider_id: meta-reference
    provider_type: inline::meta-reference
    config:
      service_name: "${env.OTEL_SERVICE_NAME:=}"
      sinks: ${env.TELEMETRY_SINKS:=console}
      sqlite_db_path: /opt/app-root/src/.llama/distributions/rh/trace_store.db
      otel_exporter_otlp_endpoint: ${env.OTEL_EXPORTER_OTLP_ENDPOINT:=}
  agents:
  - provider_id: meta-reference
    provider_type: inline::meta-reference
    config:
      persistence_store:
```

```
      type: sqlite
      namespace: null
      db_path: /opt/app-root/src/.llama/distributions/rh/agents_store.db
    responses_store:
      type: sqlite
      db_path: /opt/app-root/src/.llama/distributions/rh/responses_store.db
  vector_io:
  - provider_id: milvus
    provider_type: inline::milvus
    config:
      db_path: /opt/app-root/src/.llama/distributions/rh/milvus.db
      kvstore:
        type: sqlite
        namespace: null
        db_path: /opt/app-root/src/.llama/distributions/rh/milvus_registry.db
models:
  - model_id: llama-3-2-3b
    provider_id: vllm-inference
    model_type: llm
    metadata: {}

  - model_id: gpt-4o-mini
    provider_id: openai
    model_type: llm
    metadata: {}

server:
  port: 8321
  auth:
    provider_config:
      type: "oauth2_token"
      jwks:
        uri: "https://my-keycloak-server.com/realms/llamastack-demo/protocol/openid-
connect/certs" (1)
        key_recheck_period: 3600
      issuer: "https://my-keycloak-server.com/realms/llamastack-demo" (2)
      audience: "account"
      verify_tls: true
      claims_mapping:
        llamastack_roles: "roles" (3)
    access_policy:
      - permit: (4)
          actions: [read]
          resource: model::vllm-inference/llama-3-2-3b
        description: Allow all authenticated users to access Llama 3.2 model
      - permit: (5)
          actions: [read]
          resource: model::openai/gpt-4o-mini
        when: user with inference_max in roles
        description: Allow only users with inference_max role to access OpenAI models
```

(1) (2) Specify your Keycloak host and Realm in the URL.

(3) Maps the **llamastack_roles** path from the token to the **roles** field.

4 Policy 1: Allow all authenticated users to access vLLM models.

5 Policy 2: Restrict OpenAI models to users with the **inference_max** role.

3. Create a ConfigMap that uses the **run.yaml** configuration by running the following command:

```
$ oc create configmap llamastack-custom-config --from-file=run.yaml=run.yaml -n redhat-ods-operator
```

4. Create a **llamastack-distribution.yaml** file with the following parameters:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: llamastack-distribution
  namespace: redhat-ods-operator
spec:
  replicas: 1
  server:
    distribution:
      name: rh-dev
    containerSpec:
      name: llama-stack
      port: 8321
      env:
        # vLLM Provider Configuration
        - name: VLLM_URL
          value: "https://your-vllm-service:8000/v1"
        - name: VLLM_API_TOKEN
          value: "your-vllm-token"
        - name: VLLM_TLS_VERIFY
          value: "false"
        # OpenAI Provider Configuration
        - name: OPENAI_API_KEY
          value: "your-openai-api-key"
        - name: OPENAI_BASE_URL
          value: "https://api.openai.com/v1"
    userConfig:
      configMapName: llamastack-custom-config
      configMapNamespace: redhat-ods-operator
```

5. To apply the distribution, run the following command:

```
$ oc apply -f llamastack-distribution.yaml
```

6. Wait for the distribution to be ready by running the following command:

```
oc wait --for=jsonpath='{.status.phase}'=Ready llamastackdistribution/llamastack-distribution -n redhat-ods-operator --timeout=300s
```

7. Generate the OAuth tokens for each user account to authenticate API requests.

   a. To request a basic access token, and to add the token to a **user1.token** file, run the following command:

```
$ curl -d client_id=llamastack \
  -d client_secret=YOUR_CLIENT_SECRET \
  -d username=user1 \
  -d password=user1-password \
  -d grant_type=password \
  https://my-keycloak-server.com/realms/llamastack-demo/protocol/openid-connect/token
\
  | jq -r .access_token > user1.token
```

b. To request full access token and add it to a **user2.token** file, run the following command:

```
$ curl -d client_id=llamastack \
  -d client_secret=YOUR_CLIENT_SECRET \
  -d username=user2 \
  -d password=user2-password \
  -d grant_type=password \
  https://my-keycloak-server.com/realms/llamastack-demo/protocol/openid-connect/token
\
  | jq -r .access_token > user2.token
```

c. Verify the credentials by running the following command:

```
$ cat user2.token | cut -d . -f 2 | base64 -d 2>/dev/null | jq .
```

Example output

```
{
  "iss": "https://my-keycloak-server.com/realms/llamastack-demo",
  "aud": "account",
  "exp": 1760553504,
  "preferred_username": "user2",
  "llamastack_roles": ["inference_max"]
}
```

8. To test the Llama Stack URLs, create a route called **route.yaml** with the following example parameters:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: llamastack-distribution
  namespace: redhat-ods-operator
  labels:
    app: llamastack-distribution
spec:
  to:
    kind: Service
    name: llamastack-distribution-service
    weight: 100
  port:
    targetPort: 8321
  tls:
```

```
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
  wildcardPolicy: None
```

9. Apply the route to your cluster with the following command:

```
oc apply -f route.yaml
```

10. Verify that this route is active with the following command:

```
LLAMASTACK_URL=https://$(oc get route -n redhat-ods-operator llamastack-distribution -o
jsonpath='{.spec.host}' 2>/dev/null)
```

Verification

- Testing basic access to models
  1. Load the token with the following command:

```
USER1_TOKEN=$(cat user1.token)
```

  2. Access the vLLM model by running the following command:

```
curl -X POST ${LLAMASTACK_URL}/v1/openai/v1/chat/completions -H "Content-Type:
application/json" -H "Authorization: Bearer ${USER1_TOKEN}" -d '{"model": "vllm-
inference/llama-3-2-3b", "messages": [{"role": "user", "content": "Hello!"}], "max_tokens":
50}'
```

  3. If you attempt to access the OpenAI models without these permissions, you will see an error due to access restrictions:

```
curl -X POST ${LLAMASTACK_URL}/v1/openai/v1/chat/completions -H "Content-Type:
application/json" -H "Authorization: Bearer ${USER1_TOKEN}" -d '{"model": "openai/gpt-
4o-mini", "messages": [{"role": "user", "content": "Hello!"}], "max_tokens": 50}'
```

- Testing full authorization to models
  1. Load the token with the following command:

```
$ USER2_TOKEN=$(cat user2.token)
```

  2. Access the vLLM model by running the following command:

```
$ curl -X POST ${LLAMASTACK_URL}/v1/openai/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${USER2_TOKEN}" \
  -d '{
    "model": "vllm-inference/llama-3-2-3b",
    "messages": [{"role": "user", "content": "Hello!"}],
    "max_tokens": 50
  }'
```

  3. Access the OpenAI models by running the following command:

```
$ curl -X POST ${LLAMASTACK_URL}/v1/openai/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${USER2_TOKEN}" \
  -d '{
    "model": "openai/gpt-4o-mini",
    "messages": [{"role": "user", "content": "Hello!"}],
    "max_tokens": 50
  }'
```

- Testing without any authorization:

1. Attempt to access the OpenAI or vLLM models:

```
$ curl -X POST ${LLAMASTACK_URL}/v1/openai/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "openai/gpt-4o-mini",
    "messages": [{"role": "user", "content": "Hello!"}],
    "max_tokens": 50
  }'
```

Example output

```
{"error": {"message": "Authentication required. Please provide a valid OAuth2 Bearer token from https://my-keycloak-server.com/realms/llamastack-demo"}}
```