Red Hat OpenShift AI Cloud Service 1

Working with RAG

Working with RAG in Red Hat OpenShift AI Cloud Service

Last Updated: 2025-08-19

Red Hat OpenShift AI Cloud Service 1 Working with RAG

Working with RAG in Red Hat OpenShift AI Cloud Service

Legal Notice

Copyright © 2025 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

As a cluster administrator, you can deploy a RAG stack in Red Hat OpenShift AI.

Table of Contents

CHAPTER 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 data science project.

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.

CHAPTER 2. WORKING WITH 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 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:

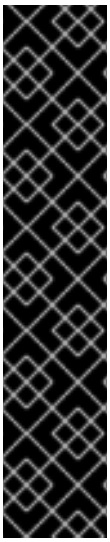
- **Inference model servers** such as vLLM, designed to efficiently serve large language models.
- **Vector storage** solutions, primarily Milvus, to store embeddings generated from your domain data.
- **Retrieval and embedding management** workflows using integrated tools, such as Docling, to handle 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 Data Science Project.

Additional resources

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

CHAPTER 3. DEPLOYING A RAG STACK IN A DATA SCIENCE PROJECT



IMPORTANT

Deploying a RAG stack in OpenShift AI is a Developer Preview feature. Developer Preview features are not supported by Red Hat in any way and are not functionally complete or production-ready. Do not use Developer Preview features for production or business-critical workloads. Developer Preview features provide early access to functionality in advance of possible inclusion in a Red Hat product offering. Customers can use these features to test functionality and provide feedback during the development process. Developer Preview features might not have any documentation, are subject to change or removal at any time, and have received limited testing. Red Hat might provide ways to submit feedback on Developer Preview features without an associated SLA.

For more information about the support scope of Red Hat Developer Preview features, see Developer Preview 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 data science 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 a data science pipeline or Jupyter notebook. This process keeps the embeddings synchronized with the source data.
- Expose and secure the model endpoints.

3.1. 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 command-line interface (CLI).

NOTE

As an alternative to following the steps in this procedure, you can activate the Llama Stack Operator from the OpenShift command-line interface (CLI) 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 cluster administrator privileges.
- You installed the OpenShift command line interface (**oc**) as described in Installing the OpenShift CLI (OpenShift Dedicated) or Installing the OpenShift CLI (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** appears and has a status of **Running**.

3.2. 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 standard deployment mode.

Prerequisites

- 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 single-model serving platform. For more information about enabling the single-model serving platform, see Enabling the single-model serving platform.
- You can access the single-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 command line interface (**oc**) as described in Installing the OpenShift CLI (OpenShift Dedicated) or Installing the OpenShift CLI (Red Hat OpenShift Service on AWS).
- You have created a data science 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 data science project.

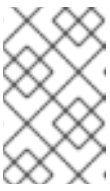
PROCEDURE

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 **Models** tab.
2. In the **Single-model serving platform** tile, click **Select single-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 **Standard** 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 command-line interface (CLI).

```
$ 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 <data science 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 <data science 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 **Models** tab on the Data Science 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 appears 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
```

3.3. 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 command line interface (**oc**) as described in Installing the OpenShift CLI (OpenShift Dedicated) or Installing the OpenShift CLI (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 command-line interface (CLI).

```
$ 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 **Model 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.

3.4. DEPLOYING A LLAMASTACKDISTRIBUTION INSTANCE

You can integrate LlamaStack and its retrieval-augmented generation (RAG) capabilities with your deployed Llama 3.2 model served by vLLM. You can use this integration to build intelligent applications that combine large language models (LLMs) with real-time data retrieval, providing more accurate and

contextually relevant responses for your AI workloads. When you create a **LlamaStackDistribution** custom resource (CR), specify **rh-dev** in the **spec.server.distribution.name** field.

Prerequisites

- 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 have 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, 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 (OpenShift Dedicated) or Installing the OpenShift CLI (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 command-line interface (CLI).

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

2. Create a secret containing 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. Log in to the OpenShift web console.
4. From the left-hand navigation, select **Administrator** view.
5. Click the **Quick Create** () icon and then click the **Import YAML** option.
6. In the **YAML** editor that appears, create a custom resource definition (CRD) similar to the following example:

```
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  name: lsd-llama-milvus
spec:
  replicas: 1
  server:
    containerSpec:
      resources:
        requests:
          cpu: "250m"
          memory: "500Mi"
        limits:
          cpu: "2"
          memory: "12Gi"
      env:
        - name: INFERENCE_MODEL
          valueFrom:
            secretKeyRef:
              key: INFERENCE_MODEL
              name: llama-stack-inference-model-secret
        - name: VLLM_URL
          valueFrom:
            secretKeyRef:
              key: VLLM_URL
              name: llama-stack-inference-model-secret
        - name: VLLM_TLS_VERIFY
          valueFrom:
            secretKeyRef:
              key: VLLM_TLS_VERIFY
              name: llama-stack-inference-model-secret
        - name: VLLM_API_TOKEN
          valueFrom:
            secretKeyRef:
              key: VLLM_API_TOKEN
              name: llama-stack-inference-model-secret
        - name: MILVUS_DB_PATH
          value: ~/.llama/milvus.db
        - name: FMS_ORCHESTRATOR_URL
          value: "http://localhost"
      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.

7. Click **Create**.

Verification

- In the left-hand navigation, click **Workloads → Pods** and then verify that the LlamaStack pod is running in the correct namespace.
- To verify that the LlamaStack 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 LlamaStack backend is present in your namespace and points to the running pod. You can check this by clicking **Networking → Services** in the web console.

3.5. 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 voids requiring a separate ingestion pipeline. By using the LlamaStack SDK, you can embed and store text in your vector store in real-time, enabling immediate RAG workflows.

Prerequisites

- You have deployed a Llama 3.2 model with a vLLM model server and you have integrated LlamaStack.
- You have created a project workbench within a data science project.
- You have opened a Jupyter notebook and it is running in your workbench environment.
- You have installed the **llama_stack_client** version 0.2.14 or later in your workbench environment.
- You have a created and configured a vector database instance and you know its identifier.

Procedure

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

```
%pip install llama_stack
```

2. In a new notebook cell, import RAGDocument and LlamaStackClient:

```
from llama_stack_client import RAGDocument, 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:

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

5. Verify that the list of registered models includes your Llama model and an embedding model. Here is an example of a list of registered models:

```
[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')]
```

6. 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 = embedding_model.metadata["embedding_dimension"]
```

7. In a new notebook cell, define the following parameters:

   - **vector_db_id**: a unique name that identifies your vector database, for example, **my_milvus_db**.

   - **provider_id**: the connector key that your Llama Stack gateway has enabled. For the Milvus vector database, this connector key is **"milvus"**. You can also list the available connectors:

```
print(client.vector_dbs.list_providers()) # lists available connectors

vector_db_id = "<your vector database ID>"
provider_id  = "<your provider ID>"
```

8. In a new notebook cell, register or confirm your vector database to store embeddings:

```
_ = client.vector_dbs.register(
vector_db_id=vector_db_id,
embedding_model=embedding_model_id,
embedding_dimension=embedding_dimension,
provider_id=provider_id,
)
print(f"Registered vector DB: {vector_db_id}")
```

IMPORTANT

If you skip this step, and as a result, you do not register your vector database with your vector database ID, an error occurs if you attempt to ingest text into your vector database.

9. In a new notebook cell, define the raw text that you want to ingest into the vector store:

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

10. In a new notebook cell, create a RAGDocument object to contain the raw text:

```
document = RAGDocument(
document_id="raw_text_001",
content=raw_text,
mime_type="text/plain",
metadata={"source": "example_passage"},
)
```

11. In a new notebook cell, ingest the raw text:

```
client.tool_runtime.rag_tool.insert(
    documents=[document],
    vector_db_id=vector_db_id,
    chunk_size_in_tokens=100,
)
print("Raw text ingested successfully")
```

12. In a new notebook cell, create a RAGDocument from an HTML source and ingest it into the vector store:

```
source = "https://www.paulgraham.com/greatwork.html"
print("rag_tool> Ingesting document:", source)

document = RAGDocument(
    document_id="document_1",
    content=source,
    mime_type="text/html",
    metadata={},
)
```

13. In a new notebook cell, ingest the content into the vector store:

```
client.tool_runtime.rag_tool.insert(
    documents=[document],
    vector_db_id=vector_db_id,
    chunk_size_in_tokens=50,
)
print("Raw text ingested successfully")
```

Verification

- Review the output to confirm successful ingestion. A typical response after ingestion includes the number of text chunks inserted and any warnings or errors.
- The model list returned by **client.models.list()** includes your Llama 3.2 model and an embedding model.

3.6. QUERYING INGESTED CONTENT IN A LLAMA MODEL

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

Prerequisites

- 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 logged in to OpenShift web console.
- 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 project workbench within a data science project.
- You have opened a Jupyter notebook and it is running in your workbench environment.
- You have installed the **llama_stack_client** version 0.2.14 or later in your workbench environment.
- You have ingested content into your model.

NOTE

This procedure does not require any specific type of content. It only requires that you have already ingested some text, HTML, or document data into your vector database, and that this content is available for retrieval. If you have previously ingested content, that content will be available to query. If you have not ingested any content yet, the queries in this procedure will return empty results or errors.

Procedure

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

```
%pip install 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. For example:

```
client = LlamaStackClient(base_url="http://lsd-llama-milvus-service:8321/")
```

4. In a new notebook cell, list the available models:

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

5. Verify that the list of registered models includes your Llama model and an embedding model. Here is an example of a list of registered models:

```
[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')]
```

6. In a new notebook cell, select the first LLM in your list of registered models:

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

7. In a new notebook cell, define the **vector_db_id**, which is a unique name that identifies your vector database, for example, **my_milvus_db**. If you do not know your vector database ID, contact an administrator.

```
vector_db_id = "<your vector database ID>"
```

8. In a new notebook cell, query the ingested content using the low-level RAG tool:

```
# Example RAG query for one-off lookups
query = "What benefits do the ingested passages provide for retrieval?"
result = client.tool_runtime.rag_tool.query(
    vector_db_ids=[vector_db_id],
```

```
    content=query,
)
print("Low-level query result:", result)
```

9. In a new notebook cell, query the ingested content by using the high-level Agent API:

```
# Create an Agent for conversational RAG queries
agent = Agent(
    client,
    model=model_id,
    instructions="You are a helpful assistant.",
    tools=[
        {
            "name": "builtin::rag/knowledge_search",
            "args": {"vector_db_ids": [vector_db_id]},
        }
    ],
)

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

# Create a session and run a streaming turn
session_id = agent.create_session("rag_session")
response = agent.create_turn(
    messages=[{"role": "user", "content": prompt}],
    session_id=session_id,
    stream=True,
)

# Log and print the agent's response
for log in AgentEventLogger().log(response):
    log.print()
```

Verification

- The notebook prints query results for both the low-level RAG tool and the high-level Agent API.
- No errors appear in the output, confirming the model can retrieve and respond to ingested content.

3.7. PREPARING DOCUMENTS WITH DOCLING FOR LLAMA STACK RETRIEVAL

You can transform your source documents with a Docling-enabled data science 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 Milvus vector database 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 instantly searchable in Llama Stack.

Prerequisites

Prerequisites

- 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 OpenShift web console.
- You have a data science project and access to pipelines in the OpenShift AI dashboard.
- You have created and configured a pipeline server within the data science 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 project workbench within a data science project.
- You have opened a Jupyter notebook and it is running in your workbench environment.
- You have installed the **llama_stack_client** version 0.2.14 or later in your workbench environment.
- You have installed local object storage buckets and created connections, as described in Adding a connection to your data science project.
- You have compiled to YAML a data science pipeline that includes a Docling transform, either one of the RAG demo samples or your own custom pipeline.
- Your data science project quota allows between 500 millicores (0.5 CPU) and 4 CPU cores for the pipeline run.
- Your data science 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 **llama_stack** client package:

```
%pip install 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:

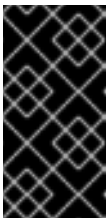
```
model_id = next(m.identifier for m in models if m.model_type == "llm")
embedding_model = next(m for m in models if m.model_type == "embedding")
embedding_model_id = embedding_model.identifier
embedding_dimension = embedding_model.metadata["embedding_dimension"]
```

6. In a new notebook cell, define the following parameters:

   - **vector_db_id**: a unique name that identifies your vector database, for example, **my_milvus_db**.
   - **provider_id**: the connector key that your Llama Stack gateway has enabled. For the Milvus vector database, this connector key is **"milvus"**. You can also list the available connectors:

```
print(client.vector_dbs.list_providers()) # lists available connectors

vector_db_id = "<your vector database ID>"
provider_id  = "<your provider ID>"
```



IMPORTANT

If you are using the sample Docling pipeline from the RAG demo repository, the pipeline registers the database automatically and you can skip this step. However, if you are using your own pipeline, you must register the database yourself.

7. In a new notebook cell, register or confirm your vector database to store embeddings:

```
_ = client.vector_dbs.register(
vector_db_id=vector_db_id,
embedding_model=embedding_model_id,
embedding_dimension=embedding_dimension,
provider_id=provider_id,
)
print(f"Registered vector DB: {vector_db_id}")
```

8. In the OpenShift web console, import the YAML file containing your docling pipeline into your data science project, as described in Importing a data science pipeline .

9. 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 database. 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_db_id**: The Milvus vector database ID.

- **service_url**: The Milvus service URL.
- **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_db_ids": [vector_db_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 database:

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

3.8. 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.

3.8.1. Supported search modes

3.8.1.1. 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.

3.8.1.2. 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.

3.8.1.3. 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.

3.8.2. 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.