SUSE

Creating SUSE AI Factory Blueprints

Creating SUSE AI Factory Blueprints

This guide walks you through creating custom blueprints for SUSE AI Factory using the command line. Learn how to quickly adapt an existing blueprint and how to build a full multi-component stack from scratch, supporting both CPU (Ollama) and NVIDIA GPU (vLLM) inference.

The instructions assume an operational Kubernetes cluster with Rancher Prime and SUSE AI Factory installed (including NVIDIA GPU support if using vLLM), and that you maintain your blueprint files in a Git repository.

Publication Date: 2026-11-12

https://documentation.suse.com

Contents

1 What is a blueprint? 1

2 What are the prerequisites? 3

2.1 Workstation requirements 3

2.2 Cluster requirements 3

3 Quick start: copy an existing blueprint 6

4 What is the blueprint directory structure? 8

5 How do I create a blueprint CR for inference with Ollama? 9

5.1 Header and metadata 9

5.2 Defining the blueprint specification 9

5.3 Selecting a chart repository 10

5.4 Configuring chart values 10

5.5 Adding an Ollama component 11

5.6 Operational considerations 12

5.7 Verifying the blueprint registration 13

6 How do I create a blueprint CR for GPU inference with vLLM? 14

6.1 Replacing Ollama with vLLM 14

6.2 Accessing the vLLM service 15

6.3 Managing gated models 16

6.4 Configuring hybrid architectures 16

7 How do I manage additional Helm repositories? 17

8 How do I document a blueprint? 18

8.1 Creating the README file 18

8.2 Providing attribution 19

9 How do I publish and run a blueprint? 20

10 How do I troubleshoot blueprint deployment issues? 22

A Blueprint templates and reference architectures 24

A1 Minimal two-component blueprint template 24

A2 ClusterRepo template 25

A3 Reference blueprints 26

1 What is a blueprint?

In SUSE AI Factory, blueprints are *custom resources* (CR) of type `Blueprint` (`ai-factory.suse.com/v1alpha1`) that declare a list of Helm charts and their configuration values. You register a blueprint on a cluster with the `kubectl apply` command.

Note

Applying the CR deploys nothing. It only *registers* the blueprint with the SUSE AI Factory Operator. Pods appear when someone creates an *AIWorkload* from that blueprint in the SUSE AI Factory UI and selects a target namespace.

A blueprint directory can contain the resource definition itself plus any other assets the stack needs. The CR `spec` has five fields:

Field	Notes
`source`	Origin of the blueprint: `SUSE`, `Nvidia`, `Partner`, `Community` or `Custom`. Use `Custom` for your own blueprints. `Community` requires SUSE AI Factory Operator 2.3.1 or newer.
`displayName`	Name displayed in the SUSE AI Factory UI.
`version`	Dotted semantic version, for example, `1.0.0`.
`description`	Free text describing what the blueprint does, prerequisites, and intended usage.
`components[]`	The list of Helm charts to install.

Each component entry contains four fields:

Field	Notes
`chartName`	Name of the chart inside the repository.
`chartRepo`	Name of a SUSE Rancher Prime `ClusterRepo` resource on the target cluster.

Field	Notes
`chartVersion`	Target version of the chart (pinned).
`values`	Verbatim Helm values for that chart.

Current limitations:

- **No namespace configuration**: The namespace is selected when creating the `AIWorkload`.
- **No release name**: Use `fullnameOverride` in `values` to ensure predictable service names when supported by the chart.
- **No dependency ordering**: Components are not sequenced. If your workload requires ordered initialization (such as populating a database before indexing), coordinate this inside the workload itself (for example, with an orchestrator DAG or init containers), rather than in the CR.

The following example illustrates a blueprint resource structure:

```
apiVersion: ai-factory.suse.com/v1alpha1
kind: Blueprint
metadata:
  name: example-blueprint
spec:
  source: Custom
  displayName: "Example Blueprint"
  version: "1.0.0"
  description: "An example blueprint definition"
  components:
  - chartName: example-chart
    chartRepo: example-repo
    chartVersion: 1.0.0
    values: {}
```

2 What are the prerequisites?

Before creating and registering custom SUSE AI Factory blueprints, ensure your workstation and target cluster satisfy the necessary prerequisites.

2.1 Workstation requirements

To author, register and inspect blueprints, you need the Kubernetes command-line tool installed and configured with administrative access to your cluster:

- **Kubernetes CLI (`kubectl`)**: Must be installed on your local workstation.

  ```
  > kubectl version --client
  ```

- **Git**: Recommended for versioning blueprint manifests, DAG files, and documentation.

2.2 Cluster requirements

Cluster prerequisites depend on whether they are required for all blueprints or only for specific workload features.

2.2.1 Mandatory requirements

Every SUSE AI Factory cluster hosting custom blueprints requires:

- **SUSE AI Factory Operator**: The operator manages the `Blueprint` and `AIWorkload` custom resource definitions (CRDs). Without it, the cluster cannot interpret blueprint manifests.
  Verify the CRD is registered:

  ```
  > kubectl get crd blueprints.ai-factory.suse.com
  ```

- **SUSE Application Collection repository (`ClusterRepo`)**: Supplies standard Helm charts packaged for SUSE AI Factory (such as Ollama, vLLM, Apache Airflow, PostgreSQL, and Grafana).

Verify the repository is registered:

```
> kubectl get clusterrepo application-collection
```

- **Repository pull credentials**: An Opaque Kubernetes secret containing authentication keys (`user` and `token`) so Helm can pull charts and container images from SUSE Application Collection. Verify the secret exists in the target operator namespace (typically `aif-operator`):

```
> kubectl -n aif-operator \
  get secret application-collection \
  -o jsonpath='{.data}' | head -c 200
```

2.2.2 Conditional requirements

Depending on the charts and features declared in your blueprint, the cluster may require additional resources:

- **Default StorageClass** (required for persistent components): Blueprints deploying databases (such as PostgreSQL or Qdrant) or caching models on disk require persistent volume claims (PVCs). If chart values do not specify an explicit storage class name, a default `StorageClass` must be defined. Verify that a default storage class is configured:

```
> kubectl get storageclass | grep '(default)'
```

- **cert-manager** (required for TLS Ingress): Needed if any blueprint component configures an external Ingress with automatic TLS certificates (for example, the Apache Airflow Web user interface). Verify that the `cert-manager` CRDs exist:

```
> kubectl get crd certificates.cert-manager.io
```

- **NVIDIA GPU support** (required for GPU inference engines like vLLM): Workloads serving models via vLLM require physical NVIDIA GPUs and the NVIDIA container runtime.
  Verify allocatable GPU capacity and the `nvidia` RuntimeClass:

```
> kubectl get nodes \
  -o custom-columns=NAME:.metadata.name,GPU:.status.allocatable.nvidia\\.com/gpu
> kubectl get runtimeclass nvidia
```

Note

Simulated GPU tools (such as kwok or fake-gpu-operator) can schedule GPU workload pods, but functional model inference requires physical GPUs.

3 Quick start: copy an existing blueprint

The fastest path to creating a blueprint is copying and adapting an existing one.

1. Download an example blueprint manifest YAML file.

```
wget https://github.com/SUSE/aif/blob/59be88cd2944b433720de3fe0c27042961f0f7e7/
charts/aif-operator/files/blueprints/simple-chatbot-with-rag-1.0.0.yaml
```

2. Edit `my-demo-ollama-1-0-0.yaml` to configure your blueprint.

Parameter	Original value	New value
CR file name	`simple-chatbot-with-rag-1.0.0.yaml`	`my-demo-olla-ma-1-0-0.yaml`
`metadata.name`	`simple-chatbot-with-rag-1-0-0`	`my-demo-ollama-1-0-0` (use hyphens)
`labels[…/blue-print-name]`	`simple-chatbot-with-rag`	`my-demo-ollama`
`spec.displayName`	`Simple Chat Bot with RAG`	`My Demo (Ollama, CPU)`
`spec.description`	Original description text	Your customized description

3. Register the blueprint on the cluster.

```
# Run a server-side dry run first to validate schema
> kubectl apply --dry-run=server -f my-demo-ollama-1-0-0.yaml

# Apply the resource to register it
> kubectl apply -f my-demo-ollama-1-0-0.yaml
> kubectl get blueprints.ai-factory.suse.com
```

4. Deploy the workload from the SUSE AI Factory Web interface.
    a. In the navigation menu, select **Blueprints**.
    b. Select **My Demo (Ollama, CPU)** and click **Create AIWorkload**.

c. Choose the target namespace.

d. Monitor the pod deployment status.

```
> kubectl -n <NAMESPACE> get pods -w
```

4 What is the blueprint directory structure?

While only the custom resource definition is mandatory, organizing files in a standardized directory structure ensures consistency across blueprints.

Partner blueprints are maintained in the SUSE partner-blueprints repository (https://github.com/SUSE/partner-blueprints/). Each blueprint should follow the directory structure described below.

```
blueprints/<ID>/
  <ID>-1-0-0.yaml        # REQUIRED — the Blueprint CR
  README.md              # Documentation and architecture details
  ATTRIBUTION.md         # Attribution details if derived from an upstream open source
 project
  clusterrepos/*.yaml    # Additional ClusterRepo definitions to apply before the CR
```

If the workload requires additional Kubernetes objects, store them in the same directory and document their deployment procedure in `README.md`.

Naming conventions:

- **Directory ID**: Matches the value in `metadata.labels[ai-factory.suse.com/blueprint-name]`.
- **CR file name**: Uses the format `<ID>-1-0-0.yaml`. The `metadata.name` field uses hyphens (for example, `my-demo-ollama-1-0-0`), while `spec.version` and the version label use dots (such as `1.0.0`).
- **Hardware variants**: Maintain CPU and GPU variants in separate sibling directories (such as `<NAME>-ollama` and `<NAME>-vllm`), using `displayName` suffixes such as `(Ollama, CPU)` and `(vLLM, GPU)` and cross-linking them in their respective `README.md` files.

5 How do I create a blueprint CR for inference with Ollama?

To create a blueprint custom resource (CR) for inference, define the metadata, blueprint specification, chart repository, and component values in a single manifest file.

5.1 Header and metadata

The metadata section defines the unique identity and labels for the blueprint.

```
apiVersion: ai-factory.suse.com/v1alpha1
kind: Blueprint
metadata:
  labels:
    ai-factory.suse.com/blueprint-name: my-demo-ollama
    ai-factory.suse.com/blueprint-version: 1.0.0
  name: my-demo-ollama-1-0-0
```

Add a comment block at the beginning of the file summarizing what the blueprint deploys, which models it serves, and any differences from sibling variants. Because `kubectl apply` preserves comments in the repository source, this provides essential context for future maintenance.

5.2 Defining the blueprint specification

The `spec` section outlines the display metadata and prerequisites.

```
spec:
  source: Custom
  displayName: My Demo (Ollama, CPU)
  version: 1.0.0
  description: |
    Deploys a demonstration stack running local models on CPU using Ollama.

    This blueprint deploys Ollama in-cluster with persistent storage while maintaining a
 lightweight resource footprint.

    Prerequisites: The SUSE AI Factory operator, the application-collection ClusterRepo
 and credentials secret, and a default StorageClass. After registering, create an
 AIWorkload from this blueprint in the Web interface.
```

Provide a concise description that ends with a clear summary of prerequisites. This description is displayed directly to users in the SUSE AI Factory Web interface.

5.3 Selecting a chart repository

The `chartRepo` parameter must reference an active SUSE Rancher Prime `ClusterRepo` resource on the target cluster. Standard repositories in SUSE AI Factory include:

Repository (`chartRepo`)	**Contents**	**Pre-registered**
`application-collection`	SUSE Application Collection charts (`ollama`, `vllm`, `apache-airflow`, `postgresql`, `grafana`)	Yes, on SUSE AI Factory clusters.
`suse-ai-registry`	SUSE AI charts (`qdrant`, `litellm`)	Yes, on SUSE AI Factory clusters.

If a chart is not in a pre-registered repository, add a `ClusterRepo` manifest in `clusterrepos/`. For more information, see *Chapter 7, How do I manage additional Helm repositories?*.

To identify available versions of a chart in the repository:

```
> kubectl get clusterrepo application-collection \
  -o yaml | grep -A2 downloadTime
> helm search repo application-collection/ollama --versions | head
```

5.4 Configuring chart values

Configure chart values under the `values` key:

```
    values:
      # Pin the service name to prevent Helm release name prefixes
      fullnameOverride: ollama
      image:
        pullPolicy: IfNotPresent
        registry: dp.apps.rancher.io
```

```
      repository: containers/ollama
      tag: 0.21.2-11.48
    persistentVolume:
      enabled: true
      accessModes:
      - ReadWriteOnce
      size: 20Gi
    service:
      port: 11434
      type: ClusterIP
    env:
    - name: QDRANT_URL
      value: http://qdrant:6333
```

Use `ClusterIP` services and forward local ports with `kubectl port-forward` during testing. Define an `ingress` block only when a component provides a Web user interface that requires external access.

5.5 Adding an Ollama component

The following configuration defines a standard Ollama component with CPU inference:

```
- chartName: ollama
  chartRepo: application-collection
  chartVersion: 1.55.0
  values:
    fullnameOverride: ollama
    image:
      pullPolicy: IfNotPresent
      registry: dp.apps.rancher.io
      repository: containers/ollama
      tag: 0.21.2-11.48
    ollama:
      gpu:
        enabled: false (1)
        draEnabled: false
      models:
        pull:
        - qwen2.5:0.5b
        run:
        - qwen2.5:0.5b
      port: 11434
    persistentVolume:
      enabled: true
      accessModes:
      - ReadWriteOnce
```

```
      size: 20Gi
    service:
      port: 11434
      type: ClusterIP
```

Key considerations for model management:

- Models listed under `models.pull` are downloaded on initial startup. For demonstrations, lightweight models such as `qwen2.5:0.5b`, `llama3.2:1b`, and `moondream:1.8b` (vision) initialize quickly.
- Models listed under `models.run` remain in memory. When serving multiple models, configure `extraEnv` variables such as `OLLAMA_KEEP_ALIVE` and `OLLAMA_MAX_LOADED_MODELS`.
- Both embeddings and chat endpoints are served by the same service (`http://ollama:11434` in-cluster).

1. Set `gpu.enabled: true` to enable GPU inference and hardware acceleration for Ollama. Once enabled you can allocate GPU resources, which will depend on the GPU vendor. Review GPU Operators (https://documentation.suse.com/cloudnative/rke2/latest/en/add-ons/gpu_operators.html) and Assign GPU resources (https://documentation.suse.com/suse-ai-factory/latest/html/AI-Factory-deployment/aif-deployment-prepare.html#ai-gpu-nodes-assigning) for additional information.

5.6 Operational considerations

Keep the following operational best practices in mind when constructing blueprints:

- **Set `fullnameOverride`**: Always define `fullnameOverride` where supported. Without it, Helm prefixes the service name with the generated release name, which breaks port-forwarding scripts and internal DNS resolution.
- **Pin generated secrets in Airflow**: When using Apache Airflow, pin the generated secret keys so that pod templates remain stable across Fleet reconciliation cycles:

```
    # Pin generated secret keys to prevent Fleet reconciliation loops
    fernetKey: "Nv3aCd-3agJbJG1QxzKMezX5pJZB4fRpZIrqhdS-MA4="
    jwtSecret: "72873f21d1682ddc023f0815344d1cdc0157e59c9c5480bcb430686d6ed466d3"
    webserverSecretKey:
"bb3ea2e3f0a3ad96a2f440ab3ca4e3406985aaf72182662ebd339f53a7a28a7a"
```

Note

For production environments, reference pre-created Kubernetes secrets rather than hardcoding static secrets in the blueprint definition.

- **Disable unused sub-charts**: When using `LocalExecutor` with Airflow, set `redis.enabled: false` and `flower.enabled: false` to avoid deploying unnecessary pods that consume cluster resources.

5.7 Verifying the blueprint registration

Validate and register your blueprint using `kubectl`:

```
> kubectl apply --dry-run=server -f my-demo-ollama-1-0-0.yaml
> kubectl apply -f my-demo-ollama-1-0-0.yaml
> kubectl get blueprint my-demo-ollama-1-0-0 -o yaml
```

6 How do I create a blueprint CR for GPU inference with vLLM?

To deploy GPU-accelerated inference with vLLM, adapt an existing CPU blueprint by creating a separate hardware variant with dedicated GPU configurations.

Hardware variants are organized in separate directories. Create a GPU variant from your CPU base configuration:

```
cp -r blueprints/my-demo-ollama blueprints/my-demo-vllm
cd blueprints/my-demo-vllm
mv my-demo-ollama-1-0-0.yaml my-demo-vllm-1-0-0.yaml
```

Update the resource fields following the renaming checklist in *Chapter 3, Quick start: copy an existing blueprint*:

1. Set `spec.displayName` to `My Demo (vLLM, GPU)`.
2. Update the `spec.description` text to specify NVIDIA GPU requirements.
3. Cross-reference the sibling CPU variant in `README.md`.

6.1 Replacing Ollama with vLLM

Replace the `Ollama` component definition with the `vLLM` component:

```
- chartName: vllm
  chartRepo: application-collection
  chartVersion: 0.1.10
  values:
    servingEngineSpec:
      runtimeClassName: nvidia
      modelSpec:
        - name: qwen25-vl-3b
          registry: dp.apps.rancher.io
          repository: containers/vllm-openai
          tag: "0.19.0-5.29"
          modelURL: Qwen/Qwen2.5-VL-3B-Instruct
          replicaCount: 1
          requestCPU: 6
          requestMemory: "16Gi"
          requestGPU: 1
```

```
            pvcStorage: "50Gi"
            vllmConfig:
              maxModelLen: 32768
              dtype: bfloat16
              gpuMemoryUtilization: 0.9
              # Vision flags: enable remote code execution and limit images per prompt
              extraArgs:
              - "--trust-remote-code"
              - "--limit-mm-per-prompt.image=2"
      routerSpec:
        enableRouter: true
        serviceType: ClusterIP
        containerPort: 8000
        servicePort: 80
```

Key configuration notes:

- `name` **and** `modelURL`: The `name` parameter provides a resource label compliant with Kubernetes naming standards. The `modelURL` parameter identifies the Hugging Face model repository to serve.
- **GPU allocation**: Each entry under `modelSpec` requests dedicated GPU resources (such as `requestGPU: 1`). Multiple models run behind a unified OpenAI-compatible routing endpoint (`vllm-router-service:80`), allowing clients to select models by name.
- **Memory management**: Tune `maxModelLen` to fit the model within GPU memory before adjusting `gpuMemoryUtilization`.
- **Common arguments**: Use `--trust-remote-code` and `--limit-mm-per-prompt.image=N` for vision models. Use `--enable-auto-tool-choice --tool-call-parser=hermes` for function calling.

6.2 Accessing the vLLM service

Test the model endpoint by forwarding the service port locally:

```
> kubectl -n <NAMESPACE> port-forward svc/vllm-router-service 8000:80
> curl -s localhost:8000/v1/models | python3 -m json.tool
```

Note

=== If the Helm release uses a custom name, the service follows the naming convention `<RELEASE_NAME>-router-service`. ===

Because vLLM provides an OpenAI-compatible API, point your client applications to `/v1/chat/completions` on the router service port.

6.3 Managing gated models

Certain models require accepting terms of use on Hugging Face before download. To provide credentials securely:

1. Configure the `hf_token` key in `modelSpec`.
2. Avoid hardcoding secret tokens directly into the blueprint CR or git repositories.
3. Reference pre-created Kubernetes secrets containing the token in production environments.

6.4 Configuring hybrid architectures

You can combine GPU and CPU inference components in the same blueprint. For example, deploy `vLLM` with GPU acceleration for large language and vision models, alongside an `Ollama` CPU instance dedicated to lightweight text embedding models (such as `nomic-embed-text`). This preserves GPU resources while maintaining high-throughput text embeddings.

7 How do I manage additional Helm repositories?

If a component requires a chart repository that is not registered on the cluster by default, create a SUSE Rancher Prime `ClusterRepo` resource and store it in `clusterrepos/`. The `metadata.name` value must match the `chartRepo` parameter in the Blueprint CR.

For example, to register an HTTPS repository:

```
apiVersion: catalog.cattle.io/v1
kind: ClusterRepo
metadata:
  name: openmetadata
spec:
  url: https://helm.open-metadata.org
```

To register an OCI registry:

```
apiVersion: catalog.cattle.io/v1
kind: ClusterRepo
metadata:
  name: bitnami
spec:
  url: oci://registry-1.docker.io/bitnamicharts
```

Apply repository resources before registering the blueprint:

```
> kubectl apply -f clusterrepos/
> kubectl get clusterrepo <REPO_NAME> -o jsonpath='{.status.conditions}'   # Wait for
 Downloaded status
> kubectl apply -f my-demo-ollama-1-0-0.yaml
```

Note upstream chart licensing in the file comment header when using non-SUSE charts.

8 How do I document a blueprint?

Documenting your SUSE AI Factory blueprint with a comprehensive `README.md` and an architecture diagram ensures that users understand its components, prerequisites and deployment process.

8.1 Creating the README file

Every blueprint must include a `README.md` providing clear instructions and architectural context. Structure the file according to the standard blueprint template:

```
# My Demo (Ollama) — SUSE AI Factory blueprint

Deploys a multi-component demo featuring **Ollama** and **Qdrant**.

Blueprint CR: [`my-demo-ollama-1-0-0.yaml`](my-demo-ollama-1-0-0.yaml)

## Architecture
![my-demo-ollama architecture diagram](../images/my-demo-ollama.png)

*Every component runs on **SUSE AI Factory** (Kubernetes / Rancher). The demo UI is shown
 as an example only and is not part of the product. Vector source: [`../images/my-demo-
ollama.svg`](../images/my-demo-ollama.svg).*

## Components
| Component | Chart (App Collection) | Role |

## How it works
## Prerequisites (on the target downstream cluster)
## 1. Publish DAGs (git-sync)
## 2. Register the blueprint
## 3. Run the workload
## 4. Run the user interface
## Customization
## Cleanup
```

8.2 Providing attribution

If a blueprint derives from an upstream open source project, add an `ATTRIBUTION.md` file detailing:

- Reused components and modifications.
- Upstream repository URL and SPDX license identifier.
- Chart and model license requirements (for example, models with non-commercial license terms).

Link to `ATTRIBUTION.md` from the lead section of `README.md`.

9 How do I publish and run a blueprint?

After creating and documenting your SUSE AI Factory blueprint, commit the files to your repository, register the custom resource on the cluster, and launch the workload from the Web interface.

1. Commit the blueprint files to a Git repository. For partner blueprints, use the SUSE partner-blueprints repository (https://github.com/SUSE/partner-blueprints/).

```
git add blueprints/my-demo-ollama
git commit -m "Add My Demo (Ollama, CPU) blueprint"
git push
```

Important

Push changes to your Git repository **before** deploying if the blueprint uses Apache Airflow: git-sync pulls DAGs directly from the pushed branch.

2. Deploy and verify the blueprint components in sequence:

   a. Apply any required custom Helm repository resources:

```
> kubectl apply -f blueprints/my-demo-ollama/clusterrepos/
```

   b. Register the blueprint resource on the cluster:

```
> kubectl apply -f blueprints/my-demo-ollama/my-demo-ollama-1-0-0.yaml
> kubectl get blueprints.ai-factory.suse.com
```

   c. In the SUSE AI Factory Web interface, navigate to **Blueprints**, select your blueprint, and click **Create AIWorkload**. Choose the target namespace.

   d. Verify that the pods and PVCs initialize correctly:

```
> kubectl -n <NAMESPACE> get pods -w
> kubectl -n <NAMESPACE> get pvc
```

e. If using Apache Airflow, unpause and execute the DAGs:

```
> kubectl -n <NAMESPACE> port-forward svc/apache-airflow-api-server 8080:8080
```

f. Forward local ports for any service that needs to communicate with the client or user interface.

3. To clean up, delete the `AIWorkload` from the SUSE AI Factory Web interface, then delete the blueprint custom resource:

```
> kubectl delete -f blueprints/my-demo-ollama/my-demo-ollama-1-0-0.yaml
```

10 How do I troubleshoot blueprint deployment issues?

The following table lists common deployment symptoms and their resolutions:

Symptom	Resolution
Ollama attempts to pull `{{MODEL}}`.	Replace template placeholder strings with literal model identifiers in the CR.
Charts fail to resolve or pods are not created.	Ensure the target `chartRepo` exists as an active `ClusterRepo`. Apply repository manifests in `clusterrepos/` and verify the `Downloaded` condition.
Airflow pods restart repeatedly without entering the `Ready` state.	Pin generated secret keys (`fernetKey`, `jwtSecret`, `webserverSecretKey`) to prevent Fleet reconciliation loops.
Port-forward or cluster DNS lookups fail.	Ensure `fullnameOverride` is specified in chart values to prevent release name prefixes in service names.
DAGs do not appear in the Airflow interface.	Verify git-sync parameters (`repo`, `subPath`, `branch`), confirm the repository is reachable from the cluster, and ensure local commits have been pushed.
DAGs appear in Airflow but do not run.	Ensure DAGs are unpaused in the Airflow user interface.
DAG import fails on third-party packages.	Package required dependencies into a customized container image.
`FileNotFoundError` on `include/` paths.	Store referenced files within `dags/`, as only `dags/` is synchronized by default.
Pods remain in `Pending` state on GPU nodes.	Verify that the NVIDIA GPU Operator is running, the `nvidia` RuntimeClass exists, and requested GPUs do not exceed allocatable cluster capacity.
PVC remains in `Pending` state.	Configure a default `StorageClass` or explicitly define an existing storage class in `values`.

Symptom	Resolution
`ImagePullBackOff` from `dp.apps.rancher.io`.	Verify the `application-collection` credentials secret. Ensure the token string contains only the raw token value without environment variable prefixes.
vLLM pods terminate with out-of-memory errors.	Reduce `maxModelLen`, then adjust `gpuMemoryUtilization`.
User interface connects but API calls fail.	Verify active `kubectl port-forward` processes and check whether environment variables point to `localhost` rather than internal cluster service addresses.

Common diagnostic commands:

```
> kubectl -n <NAMESPACE> get pods,svc,pvc
> kubectl -n <NAMESPACE> describe pod <POD_NAME>
> kubectl -n <NAMESPACE> logs deploy/<DEPLOYMENT_NAME> --tail=100
> kubectl get clusterrepo -o wide
```

A Blueprint templates and reference architectures

A1 Minimal two-component blueprint template

```
apiVersion: ai-factory.suse.com/v1alpha1
kind: Blueprint
metadata:
  labels:
    ai-factory.suse.com/blueprint-name: my-demo-ollama   # == folder name
    ai-factory.suse.com/blueprint-version: 1.0.0         # dots
  name: my-demo-ollama-1-0-0                             # dashes
# One-paragraph comment: what this demo does, which models it serves, and how it
# differs from its sibling variant.
spec:
  source: Custom  # use Community once every target operator runs 2.3.1 or newer
  displayName: My Demo (Ollama, CPU)
  version: 1.0.0
  description: |
    What the demo does and which component does what.

    What this CR deploys versus what runs locally, plus a pointer to the sibling
    variant.

    Prerequisites: the SUSE AI Factory operator, the application-collection
    ClusterRepo (+ credentials secret), a default StorageClass. After importing,
    create an AIWorkload from this blueprint in the AI Factory UI.
  components:
  # ------------------------------------------------------------------------ #
  # Ollama — local LLM, CPU. Reachable in-cluster at http://ollama:11434.
  # ------------------------------------------------------------------------ #
  - chartName: ollama
    chartRepo: application-collection
    chartVersion: 1.55.0
    values:
      fullnameOverride: ollama          # pin the service name
      image:
        pullPolicy: IfNotPresent
        registry: dp.apps.rancher.io
        repository: containers/ollama
        tag: 0.21.2-11.48
      ollama:
        gpu:
          enabled: false
          draEnabled: false
```

```
        models:
          pull:
          - qwen2.5:0.5b               # literal value — never {{MODEL}}
          run:
          - qwen2.5:0.5b
        port: 11434
      persistentVolume:
        enabled: true
        accessModes:
        - ReadWriteOnce
        size: 20Gi
      service:
        port: 11434
        type: ClusterIP
  # ---------------------------------------------------------------------- #
  # Qdrant — vector store. Reachable in-cluster at http://qdrant:6333 (REST).
  # ---------------------------------------------------------------------- #
  - chartName: qdrant
    chartRepo: suse-ai-registry
    chartVersion: 1.17.0
    values:
      fullnameOverride: qdrant
      persistence:
        accessModes: ["ReadWriteOnce"]
        size: 5Gi
      service:
        type: ClusterIP
```

A2 ClusterRepo template

```
# Registers an upstream Helm repo as a ClusterRepo, so Blueprint CR components with
# `chartRepo: <NAME>` resolve. Apply BEFORE the Blueprint CR:
#   kubectl apply -f <NAME>-clusterrepo.yaml
apiVersion: catalog.cattle.io/v1
kind: ClusterRepo
metadata:
  name: <NAME>          # MUST equal the chartRepo value used in the CR
spec:
  url: https://charts.example.com         # or: oci://registry-1.docker.io/<ORG>
```

A3 Reference blueprints

Architecture pattern	Reference blueprint
Minimal single-model CR	`visiongpt-ollama`, `visiongpt-vllm`
Full stack (Airflow, PostgreSQL, Qdrant, Ollama)	`dora-compliance-ollama`
Upstream third-party Helm repositories	`data-governance-ollama`
Custom Airflow images	`fraud-detection-ollama`
Multi-model GPU serving	`finops-multimodel-vllm`
Hybrid architecture (GPU chat and CPU embeddings)	`insurance-support-vllm`
Comprehensive README and DAG documentation	`airflow-genai-rag`