AMD EPYC

USER GUIDE
AMD EPYC 9004

RedHat® OpenShift® AI In-a-Box Solutions Build Guide

Publication	58669
Revision	1.0
Issue Date	May, 2024

Date	Version	Changes
May, 2024	1.0	Initial public release

Table of Contents

This page intentionally left blank.

Chapter 1 Overview

This user guide describes how to configure and set up OpenShift and Kubeflow on nodes powered by 4th Gen AMD EPYC processors. It focuses on deploying LLama2 models and performance benchmarking multi-user requests, specifically targeting evaluating LLMs such as llama-2-7b-chat-hf and its variant models. It includes detailed, step-by-step instructions for establishing an OpenShift multi-node cluster environment, installing Kubeflow, deploying LLama2, and conducting performance benchmarking. This user guide also also addresses the deployment process on both a single Docker container and OpenShift, along with specifying the minimum hardware and software configurations necessary for this setup.

1.1 About Large Language Models

Large Language Models (LLMs) are revolutionizing the world and reshaping how we interact with technology. Models powered by advanced artificial intelligence, such as GPT-3 and Llama2, enable breakthroughs in natural language processing by comprehending and generating human-like text. LLMs are accelerating innovation in diverse applications from crafting creative content to enhancing customer support. Their ability to understand context, generate coherent responses, and adapt to various tasks has opened new frontiers in automation, education, and research. LLMs are driving transformative change by promising a future where communication and problem-solving reach unprecedented levels of efficiency and sophistication.

LLMs offer incredible potential for automating tasks, enhancing customer experiences, and driving research, but their reliance on expensive GPUs often creates a barrier to entry, This User Guide presents a groundbreaking solution that makes cutting-edge AI accessible to any organization, regardless of budget or infrastructure limitations. This solution deploys and serves Llama2 on systems powered by 4th Gen AMD EPYC™ processors, allowing you to unlock the benefits of LLMs on your existing infrastructure. This solution offers the following benefits:

- Eliminating the need for expensive GPUs enhances access to AI technology.
- AMD EPYC 9004 Series Processors optimize Llama2 performance for fast text generation, natural language processing, and content creation.
- Easily deploy Llama2 effortlessly on the popular Red Hat® OpenShift® Container Platform (OCP) for seamless scalability, flexibility, and efficient resource management.
- Measure performance using detailed performance benchmarks.

Some of the ways you can use this solution include:

- Automating tasks to streamline workflows, boost efficiency, and empower your workforce by automating repetitive tasks, such as data analysis and report generation.
- Enhancing customer service by delivering personalized experiences through intelligent chatbots, sentiment analysis, and AI-powered recommendations.
- Driving research efforts with LLM-powered automated data analysis, document summarization, and hypothesis generation.

- Enhancing competitiveness by leveraging AI-driven opportunities for innovation across text and language-driven domains.

1.2 Technology Overview

AMD EPYC processors deliver strong AI workload performance. OCP provides a scalable foundation for deploying AI models. Kubeflow simplifies deploying and managing machine learning pipelines, while Kserve and TGI optimize model serving for fast predictions. The Llama2 LLM brings advanced natural language processing capabilities, and the OpenShift web console allows centralized management and monitoring of the entire AI infrastructure.

1.2.1 AMD EPYC Processors

4th Gen AMD EPYC processors continue to redefine the standards for modern data centers. 4th Gen AMD EPYC processors are built on the innovative x86 architecture and “Zen 4” core. 4th Gen AMD EPYC processors deliver efficient, optimized performance by combining high frequencies, the largest-available L3 cache, 128 lanes of PCIe® 5 I/O (1P), and synchronized fabric and memory clock speeds, plus support for up to 6 TB of DDR5-4800 memory. Built-in security features, such as Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV-SNP), collectively known as AMD Infinity Guard, help protect data while it is in use. (AMD Infinity Guard features vary by EPYC™ Processor generations. Infinity Guard security features must be enabled by server OEMs and/or Cloud Service Providers to operate. Check with your OEM or provider to confirm support of these features. Learn more about Infinity Guard at https://www.amd.com/en/technologies/infinity-guard. GD-183)

Some specific general purpose AMD EPYC processor models include:

- **AMD EPYC 9654:** 4th Gen processor with 96 cores and 192 threads for the most demanding applications.
- **AMD EPYC 7763:** This 3rd Gen processor balances performance and power efficiency with 64 cores and 128 threads.
- **AMD EPYC 7702:** This 2nd Gen processor is an affordable option with 64 cores and 128 threads.

Other AMD EPYC models offer from 16-128 cores to cater to specific needs and budgets.

1.3 Use Cases

This section briefly explores some compelling AI use cases specifically designed for internal and external applications within your company. Artificial intelligence (AI) is rapidly transforming business landscapes, offering exciting opportunities to enhance efficiency, improve customer experience, and drive innovation. The synergy between AMD EPYC processors and OpenShift AI presents a compelling opportunity for businesses to unlock the transformative power of artificial intelligence. The combination of exceptional performance, optimized scalability, and robust platform support empowers you to tackle diverse AI workloads efficiently and cost-effectively. Combining AMD EPYC processors and OpenShift AI provides a solid foundation for your AI journey and keeps you ahead of the curve as technology continues evolving.

1.3.1 Boost Employee Experience

- **Streamlined onboarding:** Imagine an AI assistant driving a smooth and efficient onboarding experience by guiding new hires through company policies, answering questions, and providing personalized knowledge access.
- **Instant support:** Picture employees saving time and reducing reliance on support teams by easily resolving common IT issues through intuitive AI-powered troubleshooting.
- **Seamless knowledge transfer:** Envision a system that fostering collaboration and self-service for immediate answers by seamlessly connecting employees to relevant internal knowledge bases.

1.3.2 Transform Customer Interactions

- **24/7 customer care:** Imagine an AI system ensuring consistent support around the clock by acting as your first line of defense by handling basic inquiries and freeing up human agents for complex issues.
- **Intelligent lead nurturing:** Picture an AI assistant freeing up sales reps for high-value interactions and closing deals by pre-qualifying leads, scheduling appointments, and nurturing potential customers.
- **Empowering self-service:** Envision customers readily accessing product information, resolving simple issues, and finding answers independently via an AI-powered support system.

1.3.3 Enhance Internal Collaboration

- **Effortless meeting management:** Imagine an AI assistant boosting team alignment and follow-through by automatically summarizing meetings, generating action items, and distributing those action items.
- **Enhanced project management:** Picture an AI system facilitating seamless communication and collaboration across teams by keeping track of tasks, deadlines, and resources.
- **Sparking innovation:** Envision an AI assistant unlocking your team's creative potential by fueling brainstorming sessions with insightful suggestions, capturing ideas, and documenting outcomes.

1.3.4 Red Hat OCP

Red Hat OCP simplifies deploying and managing containerized apps on Kubernetes by offering the following key features:

- Streamlined app deployment that allows you to package, deploy, and scale apps across clouds and edge.
- Automated workflows provide CI/CD integration for speedy releases.
- Built-in security and compliance features to defend your data.
- Run multiple projects within one platform using multiple tenants.

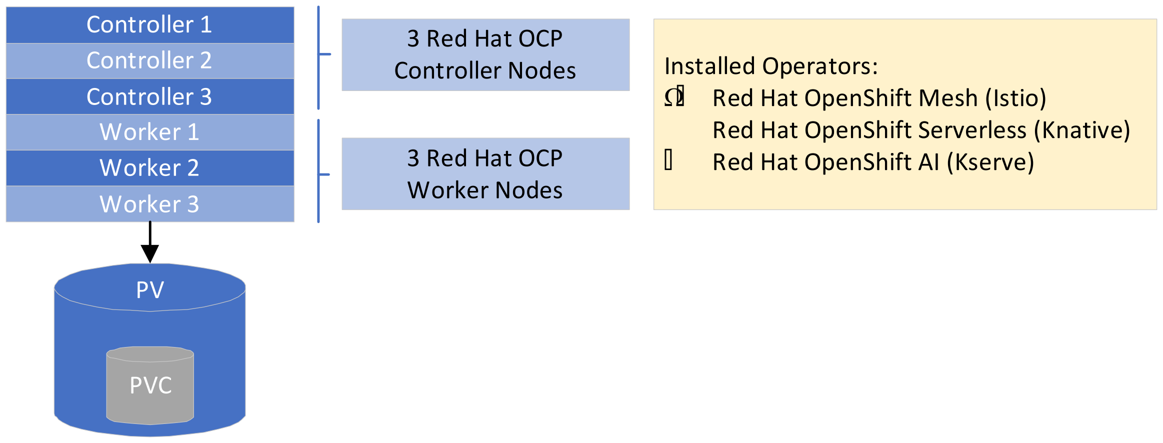

Controller 1
Controller 2
Controller 3
Worker 1
Worker 2
Worker 3
3 Red Hat OCP Controller Nodes
3 Red Hat OCP Worker Nodes
Installed Operators:
Red Hat OpenShift Mesh (Istio)
Red Hat OpenShift Serverless (Knative)
Red Hat OpenShift AI (Kserve)
PV
PVC

Figure 1-1: Red Hat OCP AI architecture

1.3.5 Kubeflow

Kubeflow is an open-source platform that seamlessly integrates with Kubernetes to simplify deploying and managing machine learning (ML) workflows. Kubeflow allows you to:

- Mix and match modular components for notebooks, data processing, and model serving.
- Rapidly deploy models on your existing Kubernetes setup with no infrastructure hurdles.
- Seamlessly scale to handle large datasets and complex algorithms.
- Obtain consistent training and deployment across environments.
- Optimize costs.

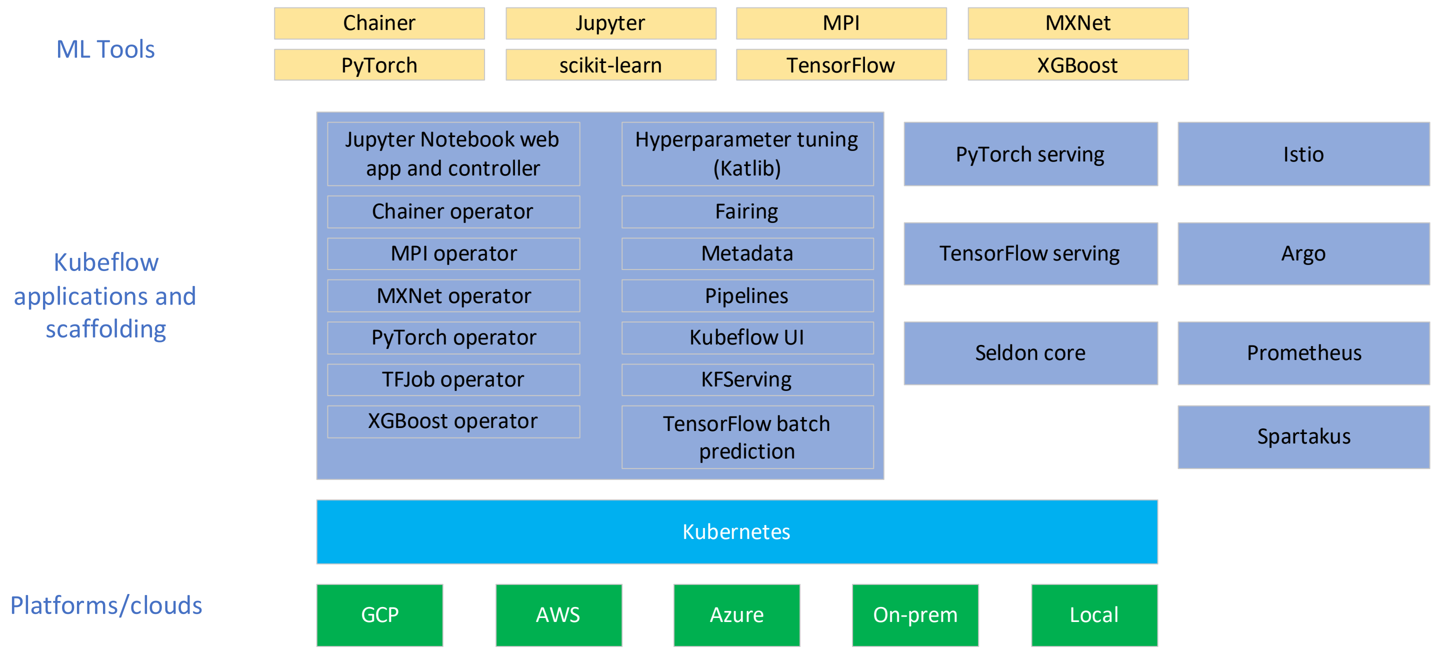


Figure 1-2: Kubeflow Architecture

1.3.6 KServe

KServe connects machine learning models to the real world by simplifying deploying and serving trained models on Kubernetes. Key KServe features include:

- Auto scaling that adjusts the number of model replicas to meet fluctuating demands.
- Service mesh with CPU/GPU/TPU support, model mesh support, and support for multiple deep learning model frameworks.
- Standardized, framework-agnostic way to serve LLMs.

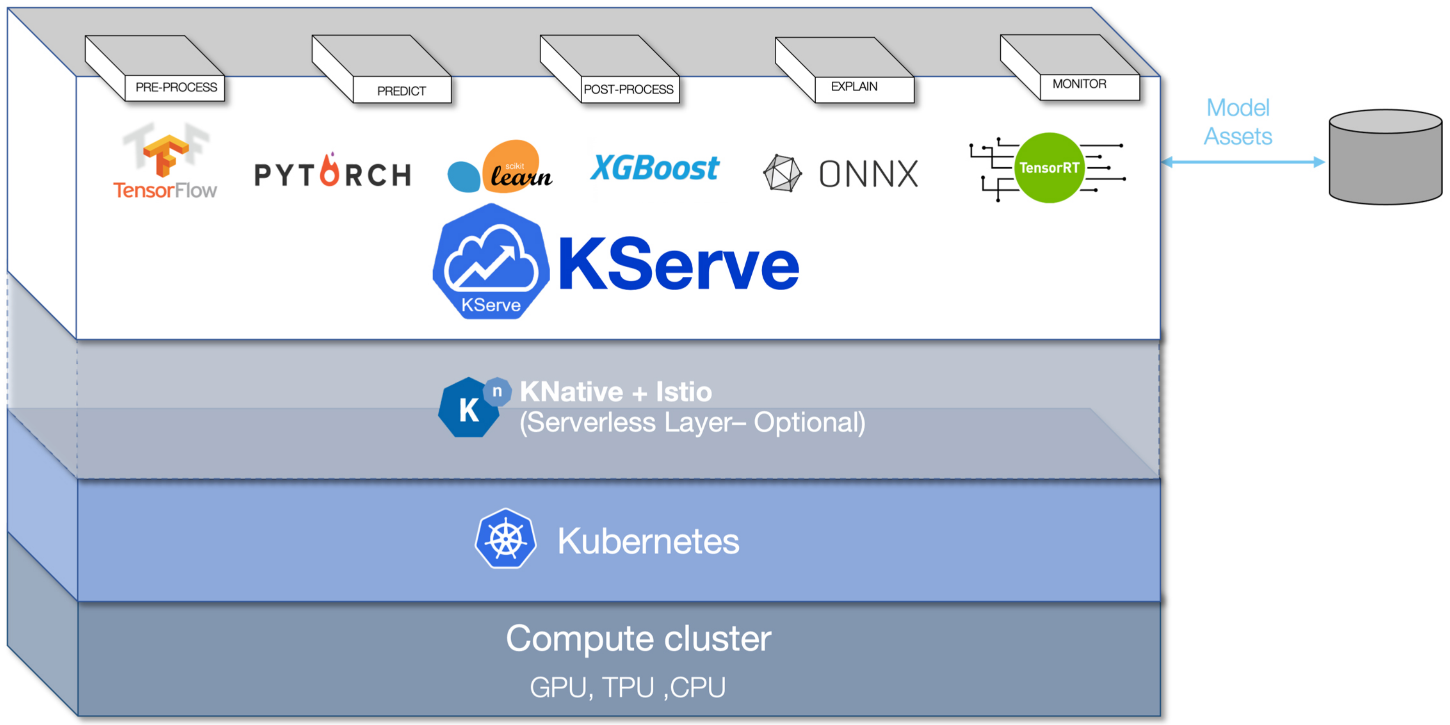


Figure 1-3: KServe architecture

1.3.7 Knative

Knative is an open-source platforme that extends Kubernetes functionality to enable building, deploying, and managing serverless and event-driven applications. It eliminates server management by automating resource provisioning, scaling, and application lifecycles. Knative's portability allows running workloads across various platforms with Kubernetes, while its focus on developer experience simplifies development through tools and abstractions. The platform consists of three core components:

- Build automates building container images.
- Serving manages deployment, scaling, and routing.
- Eventing facilitates event-driven communication within applications.

1.3.8 Istio

Istio is an open-source service mesh platform that provides capabilities for managing and securing microservices-based applications. It features traffic management, load balancing, service-to-service authentication and authorization, telemetry, and observability. Istio deploys a sidecar proxy alongside each service instance in the mesh, allowing it to intercept and control all inbound and outbound traffic to the service. This enables advanced traffic management features like traffic routing, fault injection, and circuit breaking. Istio helps in making microservices communication more reliable, secure, and observable by providing a unified control plane for managing service-to-service communication within a distributed application architecture.

1.3.9 Text Generation Inference

Text Generation Inference (TGI) is a toolkit that deploys and serves pre-trained LLMs for high-performance text generation. It supports popular open-source LLMs, including Llama, Falcon and more and can be applied to diverse use cases, such as:

- Automatic content generation, such as summaries, news articles, or marketing copy at scale.
- Create personalized adaptive or interactive text-based experiences like chatbots or simulations.
- Enhance creativity by exploring new ideas and writing styles with AI-assisted brainstorming.

Some key TGI features include:

- Tensor parallelism for faster inference on multi-GPU systems.
- Token streaming using Server-Sent Events (SSE).
- Continuous batching of incoming requests for increased total throughput.
- Optimized transformers code for inference using Flash Attention* and Paged Attention* on the most popular architectures.
- Quantization with bitsandbytes* and GPT-Q*.
- Safetensors* weight loading.
- Watermarking with A Watermark for Large Language Models*.
- Logits warper (temperature scaling, top-p, top-k, repetition penalty (see transformers.LogitsProcessor*).
- Stop sequences.
- Log probabilities.
- Custom prompt generation.
- Fine-tuning support.

TGI uses a three-step process:

1. You input text prompts, keywords, or specific instructions to the LLM.
2. The LLM uses its learned knowledge to predict the next word or text sequence based on your input (inference).
3. The inference tool gathers the generated text and presents it to you in a readable format.

1.3.10 Single Model Serving Platform

OCP offers a comprehensive solution for deploying LLMs via a Single Model Serving Platform (SMSP) based on the KServe component. This platform streamlines deploying, monitoring, scaling, and maintaining LLMs. This SMSP consists of the following components:

- **KServe:** Kubernetes Custom Resource Definition (CRD) that orchestrates model serving for all types of models. It includes model-serving runtimes that implement loading given types of model servers. KServe handles the deployment object, storage access, and network setup.
- **Red Hat OpenShift Serverless:** Cloud-native development model that allows server-less model deployments. OpenShift Serverless is based on the open-source Knative project.
- **Red Hat OpenShift Service Mesh:** Service mesh networking layer that manages traffic flows and enforces access policies. OpenShift Service Mesh is based on the open-source Istio project.

The SMSP can be deployed in several ways:

- Automated Installation
- Manual Installation
- Installing KServe Dependencies

1.3.11 TGIS Inference Server with Caikit

The Text Generation Inference Server (TGIS) with Caikit offers an efficient solution for deploying LLMs and conducting text generation tasks via a user-friendly interface that allows configuring and deploying LLMs with no need for extensive technical knowledge or manual setup. Caikit is an AI toolkit that uses developer-friendly APIs to manage models. It provides a consistent format for creating and using AI models against a wide variety of data domains and tasks. It acts as a wrapper that oversees the TGIS process, furnishes endpoints for inference, and incorporates modules tailored for diverse model types. The Caikit-nlp extension specializes in managing Natural Language Processing (NLP) models.

1.3.12 Llama2 LLM

A LLM is a deep learning algorithm that performs a variety of NLP tasks. LLMs use transformer models that use self-attention mechanism to process sequential data and analyze relationships between different elements in the sequence without relying on recurrent networks. LLMs are trained using huge datasets.

Llama2 is one such open-source, publicly available LLM. It is an advanced AI platform that combines cutting-edge algorithms, extensive data sets, and powerful computational capabilities to deliver exceptional results. Its versatile machine learning models excel in domains from NLP to image recognition. Llama2 prioritizes privacy and security to protect confidential user data. Llama2 also fosters collaboration by seamlessly integrating with other AI frameworks.

For enterprises, Llama2 can enhance customer service by providing intelligent chatbots capable of understanding and responding to user inquiries. It can automate data analysis, allowing businesses to extract valuable insights, optimize processes, and make informed decisions. Llama2's NLP capabilities can facilitate sentiment analysis, brand monitoring, and social media listening, thereby helping enterprises understand customer sentiment and market trends. Llama2 enables personalization and recommendation systems to enhance customer experiences and drive sales.

Llama2 is available in the following editions based on the number of included parameters:

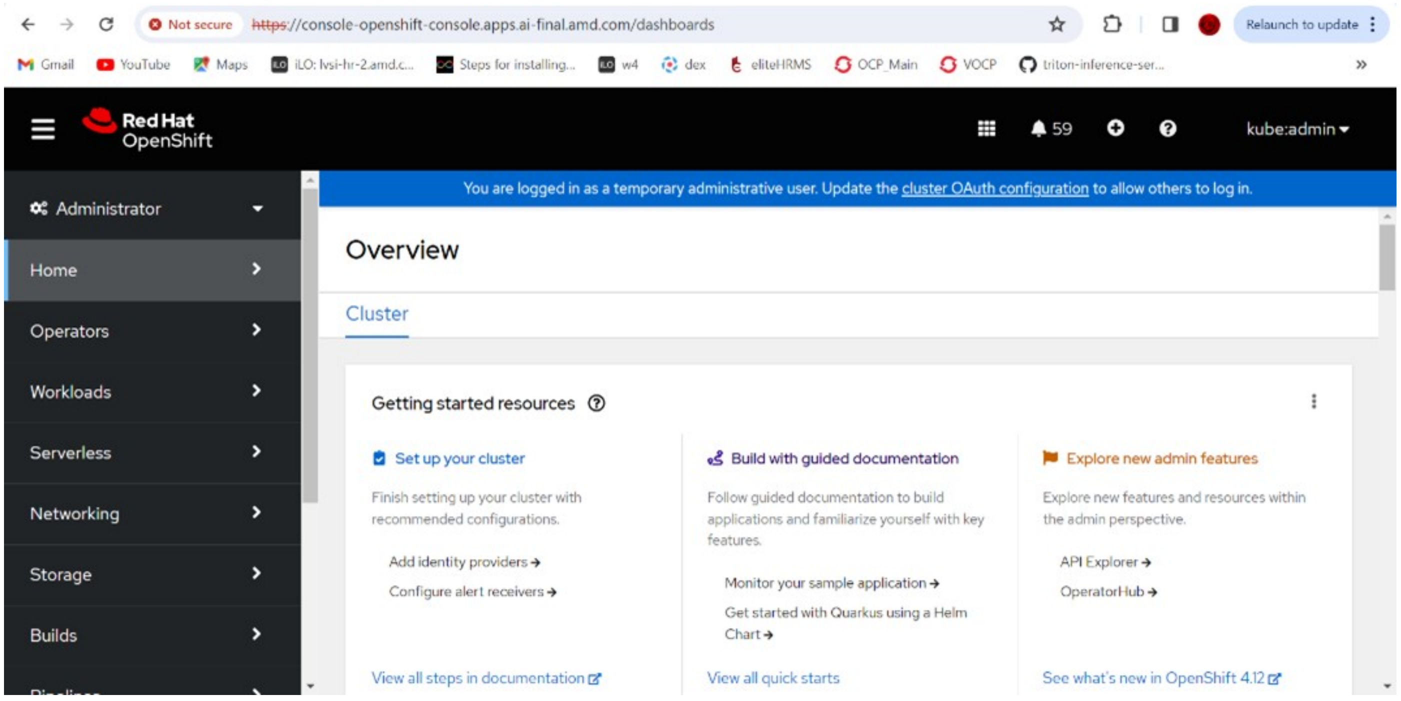
- Llama2 7B combines a substantial 7 billion-parameter model with advanced algorithms that enable generating coherent, high-quality text across different tasks. It strikes a balance between efficiency and performance, making it a versatile choice for a range of applications.

- Llama2 13B uses a 13 billion-parameter model that excels in producing nuanced and contextually rich text that is suited for complex content generation and language-related tasks.
- Llama2 70B uses a 70 billion-parameter model that pushes the boundaries of text generation to deliver exceptional fluency, coherence, and domain expertise. It is ideal for data-intensive projects or those requiring the utmost precision and scale.

1.3.13 OpenShift Web Console

The OpenShift Web Console is a web-based user interface that allows you to manage applications, resources, and projects within your OCP environment. This user-friendly control panel simplifies tasks. such as:

- **Deploying apps:** Easily upload code, configure deployments, and launch your applications. Streamline tasks and actions using integrated automation capabilities.
- **Monitoring health:** Quickly track resource usage, application health, and identify potential issues. Use clear visuals to track key metrics and gain insights into your OCP environment.
- **Managing resources:** Allocate resources to projects, scale deployments, and ensure efficient utilization. Manage multiple projects and teams within one centralized location.
- **Controlling access:** Assign roles and permissions to users and teams to enforce security and governance.
- **Extending and customizing functionality:** Use plugins and extensions based on your specific needs.


Not secure
https://console-openshift-console.apps.ai-final.amd.com/dashboards
Relaunch to update
Red Hat OpenShift
59
kube:admin
Administrator
Home
Operators
Workloads
Serverless
Networking
Storage
Builds
Overview
Cluster
Getting started resources
Set up your cluster
Add identity providers
Configure alert receivers
View all steps in documentation
Build with guided documentation
Monitor your sample application
Get started with Quarkus using a Helm Chart
View all quick starts
Explore new admin features
API Explorer
OperatorHub
See what's new in OpenShift 4.12

You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Finish setting up your cluster with recommended configurations.

Follow guided documentation to build applications and familiarize yourself with key features.

Explore new features and resources within the admin perspective.

Figure 1-4: OpenShift web console

Chapter 2

Solution Design

2.1 Node Roles

The solution described in this User Guide requires the following node types:

- **Client:** Client nodes acts as the interface between the user and the OpenShift cluster and can be used to access, get information, perform operations, and deploy applications and services on an OpenShift cluster.
- **Load Balancer:** Load balancers distribute load/requests coming in to the OpenShift Cluster from the Client across the available Worker nodes.
- **Control Plane:** Nodes that provide instructions to the worker nodes for the functions they need to perform and run services required to control an OpenShift cluster.
- **Workers:** Nodes that run workloads or containerized applications.

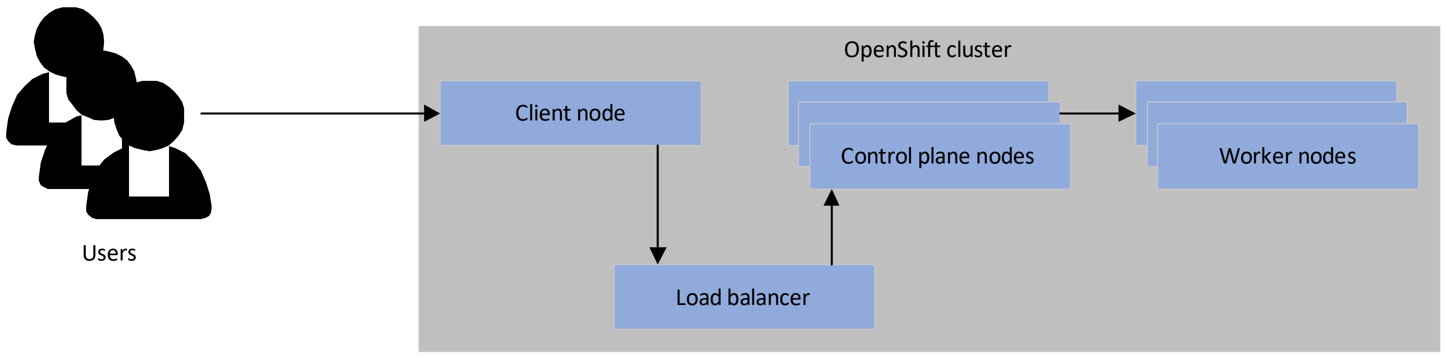


Figure 2-1: OpenShift cluster topology

2.2 Hardware Prerequisites

An OpenShift multi-node cluster must meet the hardware requirements shown in Table 2-1.

Minimum OpenShift Requirements	
Control Plane Nodes	
# of Nodes	3 (for multi-node clusters)
CPU Cores	4
RAM	16 GB
Storage	100 GB
Worker Nodes	
CPU Cores	2
RAM	8 GB
Storage	100 GB
Networking	
See Installing OpenShift Container Platform with the Assisted Installer*.	

Table 2-1: Minimum OpenShift requirements

2.3 Hardware/Software Configurations

The sample Red Hat OpenShift Container Platform and Kubeflow serving Llama2 model using TGI inference Server deployment described in this User Guide use the following hardware and software configurations:

Hardware	Configuration
CPUs	2 x AMD EPYC 9654
CPU cores | threads	96 | 192 per processor
RAM	256 GB
Storage	16 GB NVMe
Networking	2 x 100 Gbps
	8 GB
Storage	100 GB

Table 2-2: Sample hardware configuration

Software	Version
Red Hat OpenShift Container Platform	4.12
Kubeflow	1.8
Docker	1.6
Llama2-chat-hf	7B, 13B, and 70B
PyTorch	1.1.3

Table 2-3: Sample software configuration

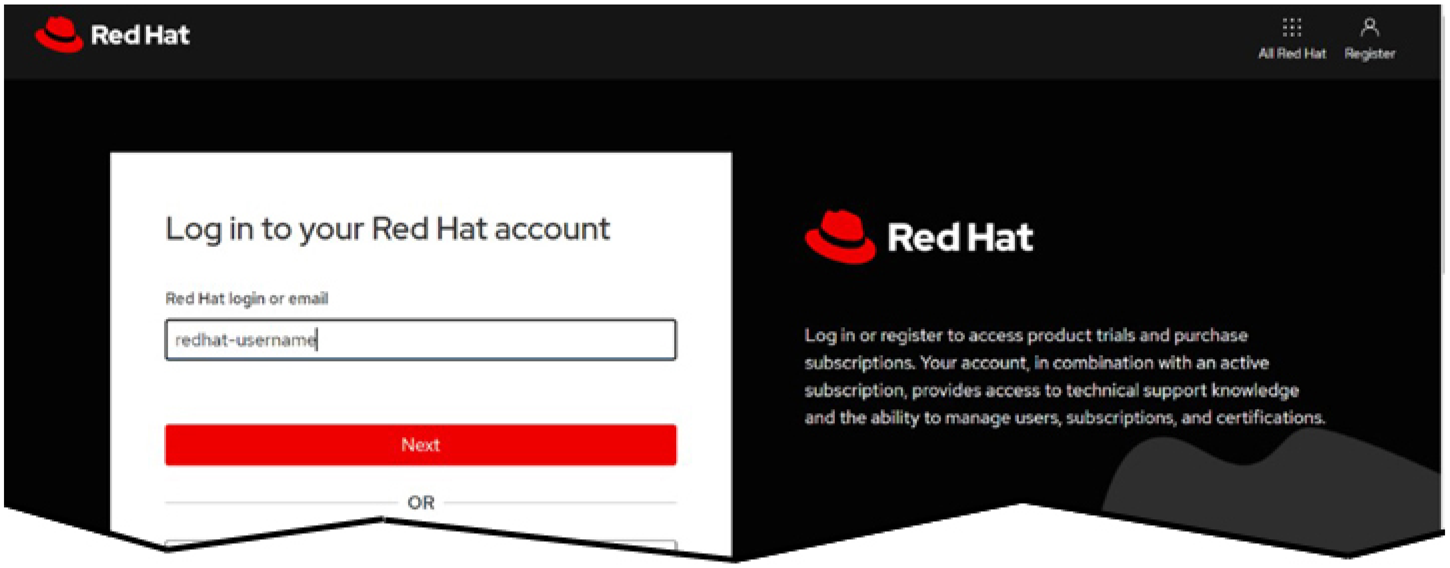
Chapter 3 Installation

This chapter contains detailed instructions for creating a multi-node OpenShift cluster using the Assisted Installer for OpenShift Container Platform utility available from Red Hat.

3.1 Deploying an OpenShift Cluster Using the Assisted Installer

These instructions are based on OpenShift 4.12.0.

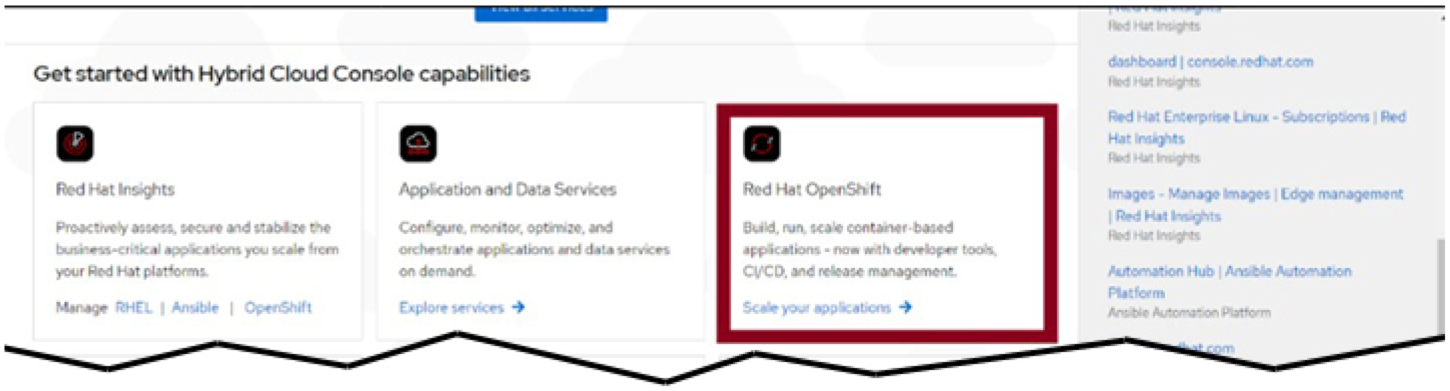
1. Login to the Red Hat Hybrid Cloud Console.



Log in or register to access product trials and purchase subscriptions. Your account, in combination with an active subscription, provides access to technical support knowledge and the ability to manage users, subscriptions, and certifications.

Figure 3-1: OpenShift Cloud Console login screen

The **Get started with Hybrid Cloud Console capabilities** screen appears.


Get started with Hybrid Cloud Console capabilities

Red Hat Insights
Proactively assess, secure and stabilize the business-critical applications you scale from your Red Hat platforms.
Manage RHEL | Ansible | OpenShift

Application and Data Services
Configure, monitor, optimize, and orchestrate applications and data services on demand.
Explore services

Red Hat OpenShift
Build, run, scale container-based applications - now with developer tools, CI/CD, and release management.
Scale your applications

Red Hat Insights
dashboard | console.redhat.com
Red Hat Insights
Red Hat Enterprise Linux - Subscriptions | Red Hat Insights
Red Hat Insights
Images - Manage Images | Edge management | Red Hat Insights
Red Hat Insights
Automation Hub | Ansible Automation Platform
Ansible Automation Platform

Figure 3-2: Get started with Hybrid Cloud Console capabilities screen

2. In the **Red Hat OpenShift** tile, click **Scale your applications**.

   The **Overview** screen appears.

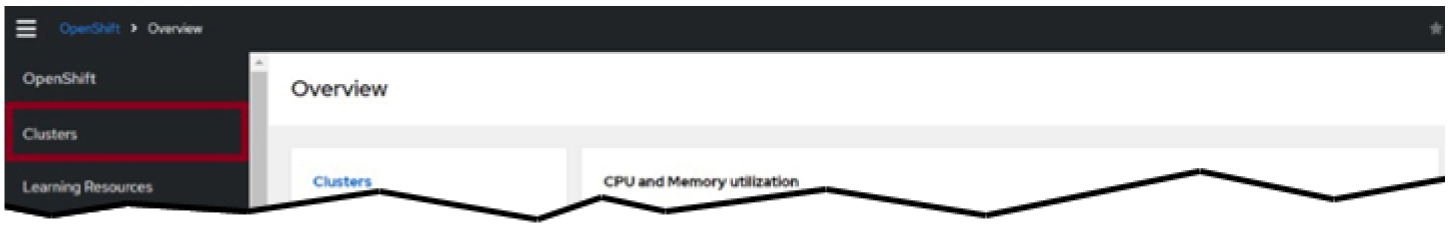

OpenShift > Overview
OpenShift
Clusters
Learning Resources
Overview
Clusters
CPU and Memory utilization

Figure 3-3: Overview screen.

3. In the left-hand menu, click **Clusters**.

   The **Clusters** screen appears.

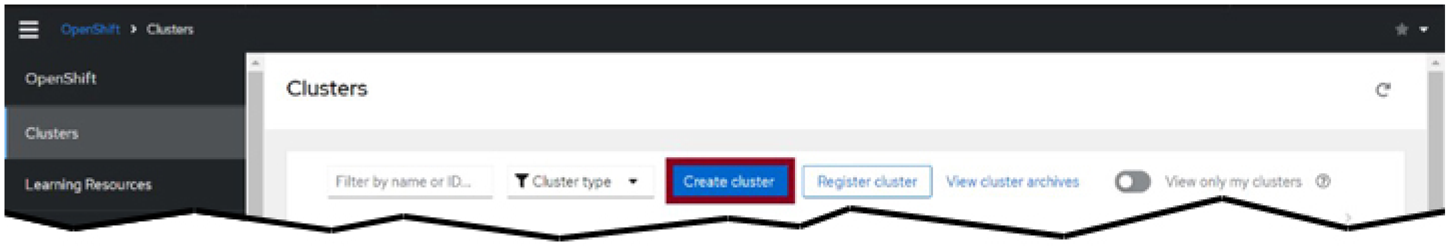

OpenShift > Clusters
OpenShift
Clusters
Learning Resources
Clusters
Filter by name or ID...
Cluster type
Create cluster
Register cluster
View cluster archives
View only my clusters

Figure 3-4: Clusters screen

4. Click **Create Cluster**.

   The **Select an OpenShift cluster type to create** screen appears.

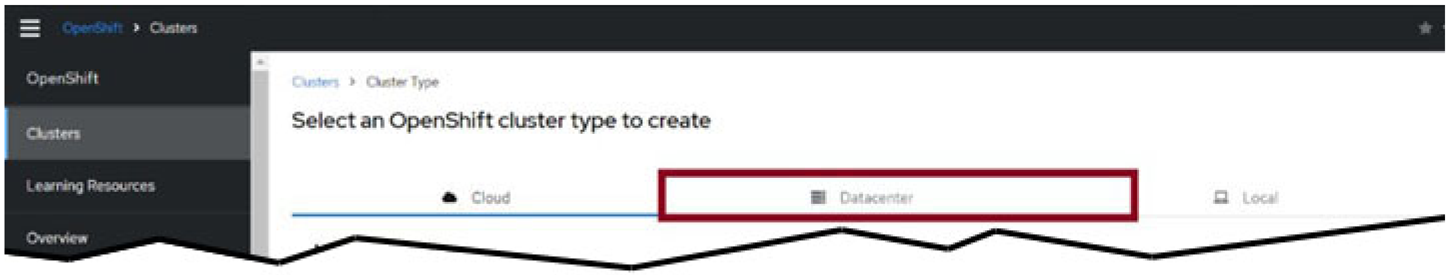
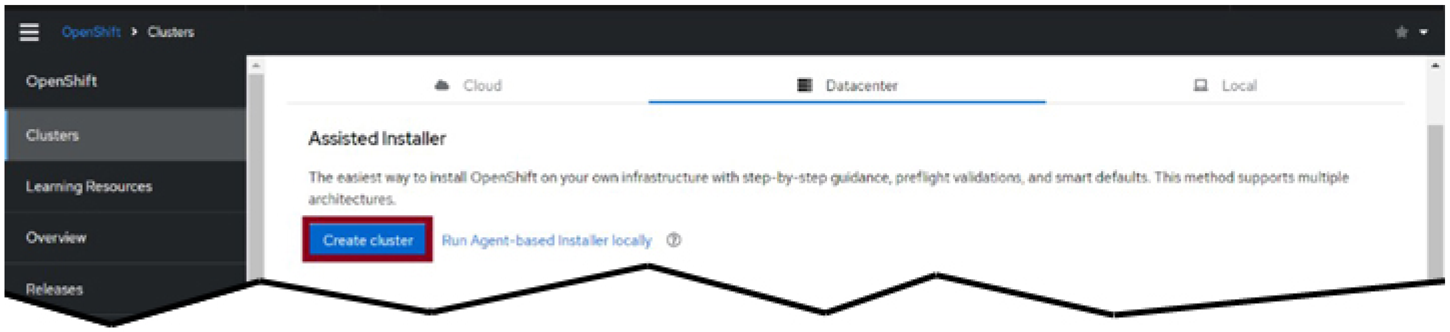


Figure 3-5: Select an OpenShift cluster type to create screen.

5. Select **Datacenter**.

   The **Assisted Installer** screen appears.

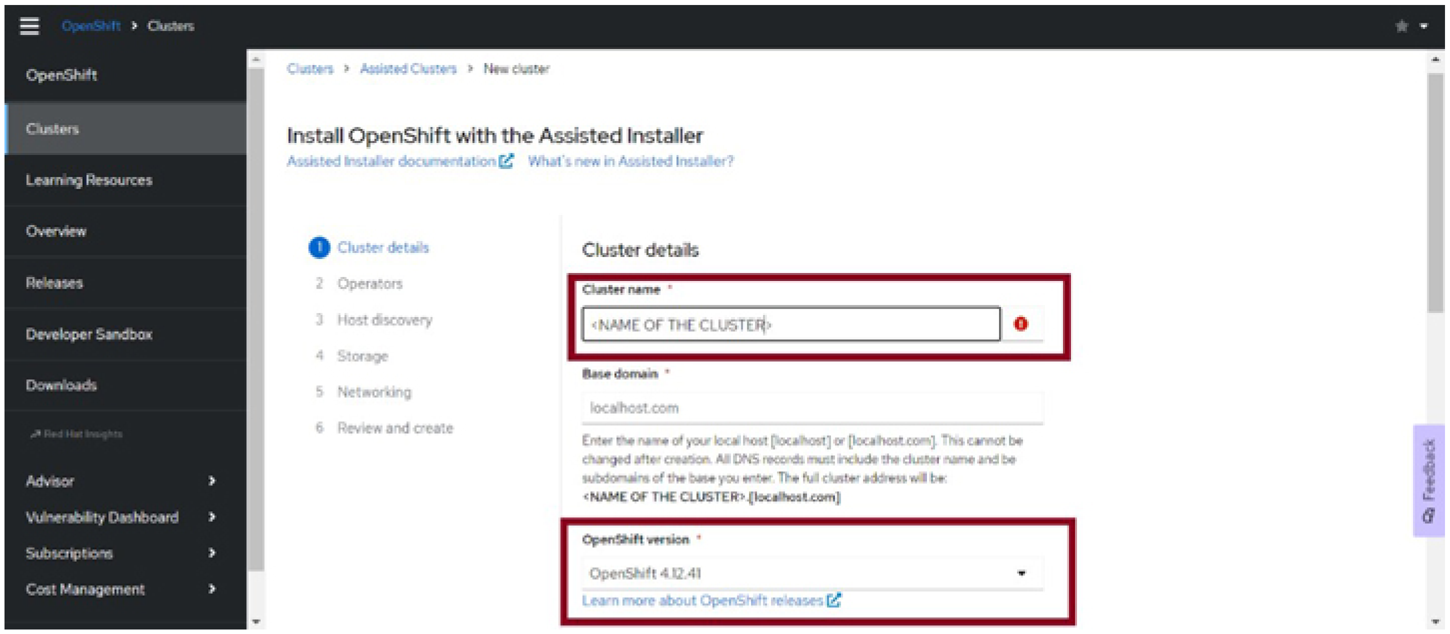


The easiest way to install OpenShift on your own infrastructure with step-by-step guidance, preflight validations, and smart defaults. This method supports multiple architectures.

Figure 3-6: Assisted Installer screen

6. Click **Create cluster**.

   The **Install OpenShift with the Assisted Installer** screen appears.



Install OpenShift with the Assisted Installer

Cluster details

Cluster name *

<NAME OF THE CLUSTER>

Base domain *

localhost.com

Enter the name of your local host [localhost] or [localhost.com]. This cannot be changed after creation. All DNS records must include the cluster name and be subdomains of the base you enter. The full cluster address will be: <NAME OF THE CLUSTER>.[localhost.com]

OpenShift version *

OpenShift 4.12.41

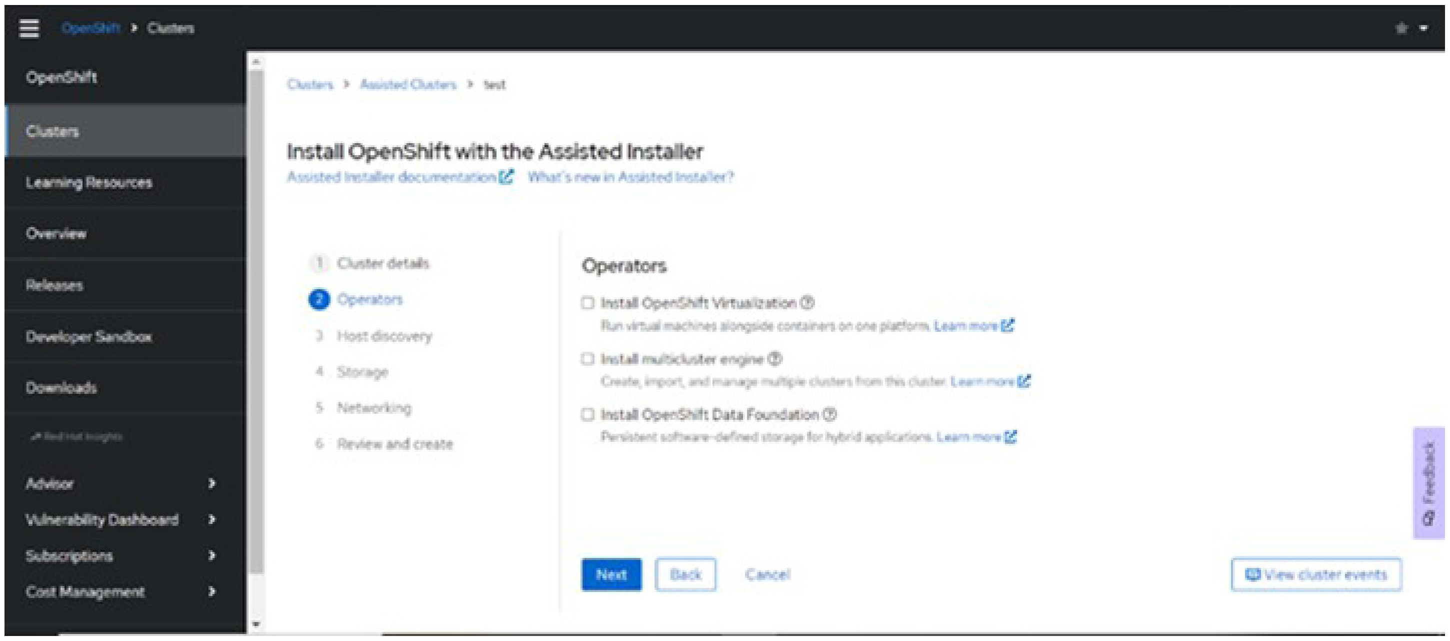
Figure 3-7: Install OpenShift with the Assisted Installer screen

7. Provide the following information:

   - **Cluster name:** Specify a name for your cluster.
   - **Base domain:** Provide a base domain (e.g., example.com). All DNS records will be named `<cluster-name>.example.com`. You cannot change the base domain after cluster installation.
   - **OpenShift version:** Select the OpenShift version to use (e.g., 4.12.0).

8. Click **Next**.

   The **Operators** screen appears.



Run virtual machines alongside containers on one platform. Learn more

Create, import, and manage multiple clusters from this cluster. Learn more

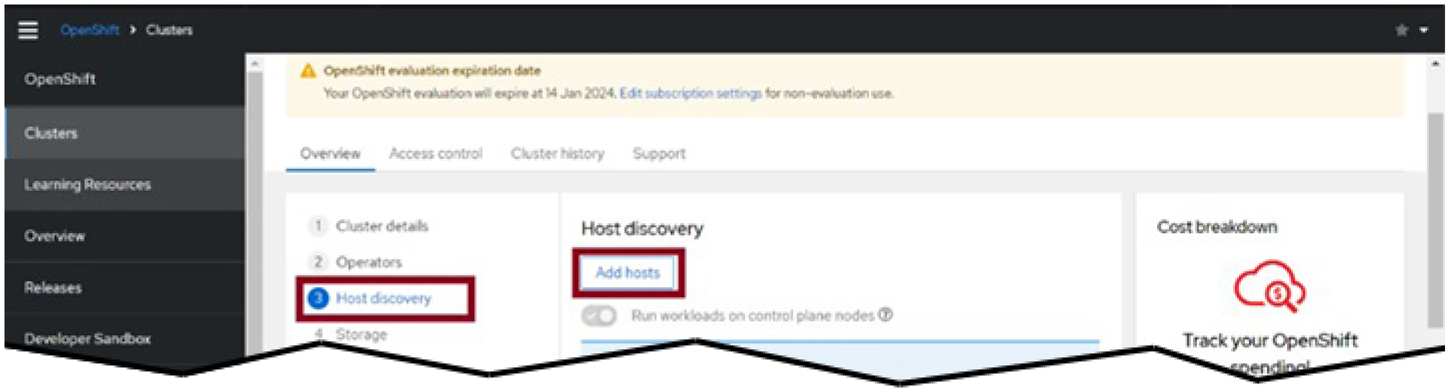
Persistent software-defined storage for hybrid applications. Learn more

Figure 3-8: Operators screen

9. If desired, you can install any operators that your cluster may require. You can also install operators after creating the cluster. This step is optional.

10. Click **Next**.

    The **Host Discovery** screen appears. You must add at least three control plane hosts. Hosts are added by booting into a Discovery ISO that can be generated using the Assisted Installer. This ISO runs Red Hat Enterprise Linux Core OS (RHCOS).



OpenShift evaluation expiration date

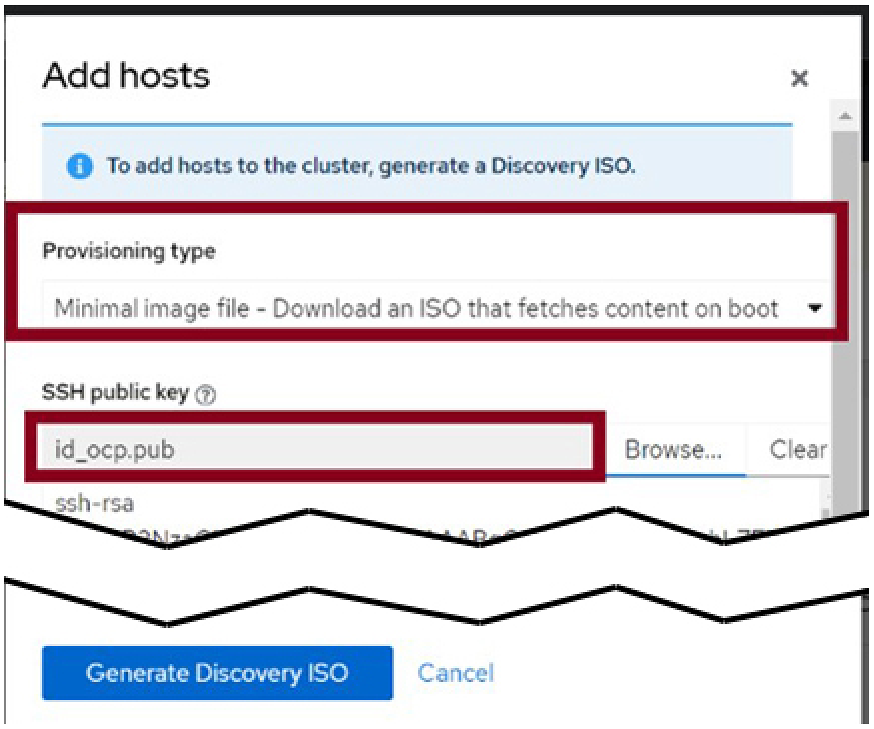
Your OpenShift evaluation will expire at 14 Jan 2024. Edit subscription settings for non-evaluation use.

Figure 3-9: Host Discovery screen

11. Click the **Add hosts** button.

    The **Add hosts** screen appears.

    Figure 3-10: Add hosts screen (right)



Add hosts

To add hosts to the cluster, generate a Discovery ISO.

Provisioning type: Minimal image file - Download an ISO that fetches content on boot

SSH public key: id_ocp.pub

ssh-rsa

12. Provide the following information:

    - **Provisioning typer:** Select **Minimal image file - Download an ISO that fetches content on host**. This option provision hosts using virtual media to download a smaller image that will fetch the data needed to boot. The nodes must have virtual media capability. AMD recommends this for both x86_64 and arm64 processor architectures.

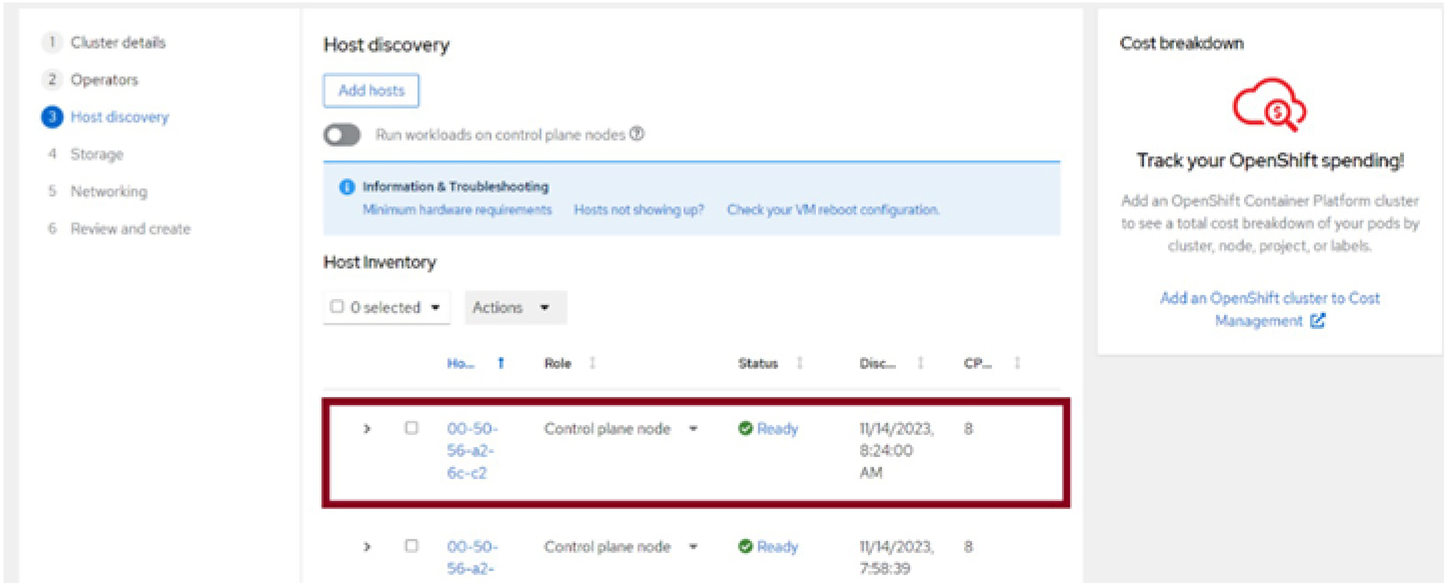
    - **SSH public key:** Add a SSH public key to enable connecting to these node as the root user. Being able to login to the cluster nodes can provide you with debugging information during installation.

13. Click **Generate Discovery ISO**.

14. Use the Discovery ISO file or URL as virtual media, then boot into RHCOS from every host that is part of your cluster.

    Each node will appear in the Red Hat Assisted Installer Console as it boots into the RHCOS. It may take 1-2 minutes for the console to discover a host post-boot.

15. Wait until all nodes appear in the Assisted Installer Console and show a **Status** of **Ready**.



1 Cluster details
2 Operators
3 Host discovery
4 Storage
5 Networking
6 Review and create
Host discovery
Add hosts
Run workloads on control plane nodes
Information & Troubleshooting
Minimum hardware requirements
Hosts not showing up?
Check your VM reboot configuration
Host Inventory
0 selected
Actions
Cost breakdown
Track your OpenShift spending!
Add an OpenShift Container Platform cluster to see a total cost breakdown of your pods by cluster, node, project, or labels.
Add an OpenShift cluster to Cost Management

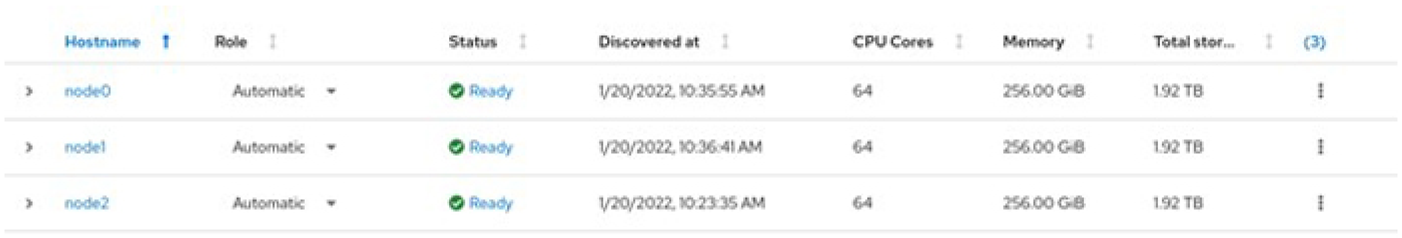
Ho...	Role	Status	Disc...	CP...
00-50-56-a2-6c-c2	Control plane node	Ready	11/14/2023, 8:24:00 AM	8
00-50-56-a2-	Control plane node	Ready	11/14/2023, 7:58:39	8

Figure 3-11: Host inventory

16. For each host:
    - Click the host MAC address.
    - Enter a name for the host.
    - Use the **Role** pull-down menu to assign a role to the selected node (**Control Plane Node** or **Worker Node**).
    - Use the following **Networking** settings:
        > **Networking management type:** Select **Cluster-managed Networking**.

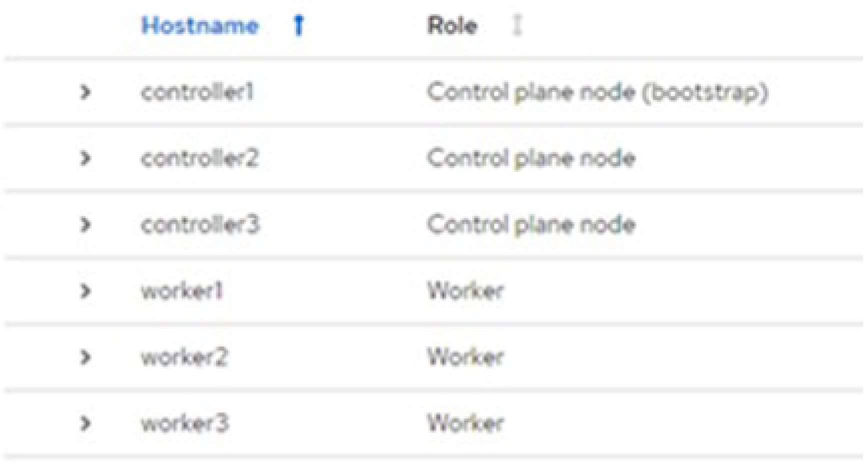
        > **Stack type:** Select **IPv4**.
    - Use the following **Advanced Networking Settings**:
        > **Networking type:** Select **Software-Defined Networking (SDN)**.

        > Check the **Automatically assign Cluster API IP and Ingress IP** check box.



	Hostname	Role	Status	Discovered at	CPU Cores	Memory	Total stor...	(3)
›	node0	Automatic	Ready	1/20/2022, 10:35:55 AM	64	256.00 GiB	1.92 TB	⋮
›	node1	Automatic	Ready	1/20/2022, 10:36:41 AM	64	256.00 GiB	1.92 TB	⋮
›	node2	Automatic	Ready	1/20/2022, 10:23:35 AM	64	256.00 GiB	1.92 TB	⋮

Figure 3-12: All nodes ready



	Hostname	Role
›	controller1	Control plane node (bootstrap)
›	controller2	Control plane node
›	controller3	Control plane node
›	worker1	Worker
›	worker2	Worker
›	worker3	Worker

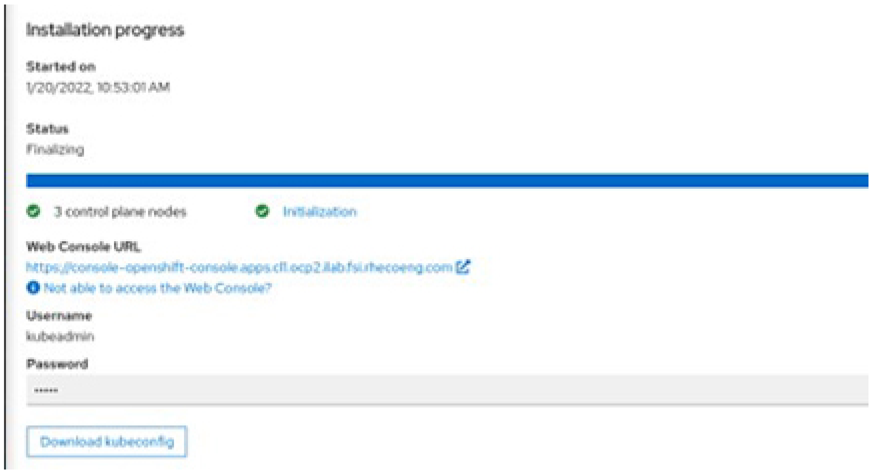
Figure 3-13: Assigned node roles

17. Select **Install Cluster**.

18. During installation, obtain the following from the Installation progress screen:

    Figure 3-14: Obtaining cluster information (right)

    - Cluster login credentials.
    - Kubeconfig file. Download and save this file; you will need it on a client machine in order to connect the OpenShift CLI Client to the installed cluster.



Installation progress

Started on
1/20/2022, 10:53:01 AM

Status
Finalizing

3 control plane nodes
Initialization

Web Console URL
https://console-openshift-console.apps.cl1.ocp2.ilab.fsi.rhecoeng.com
Not able to access the Web Console?

Username
kubeadmin

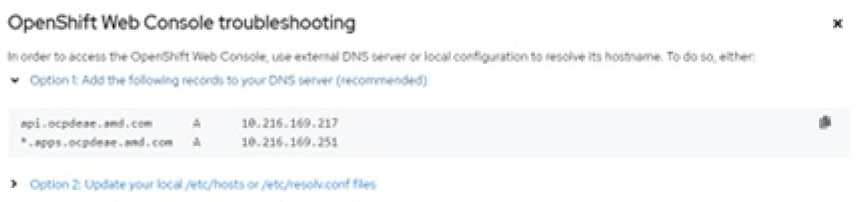
Password
*****

Download kubeconfig

19. Wait for the cluster to finish installing.

    One or more host(s) may encounter an issue where the host **Status** reads **Pending User Action**. If this happens, you can check the message and download the installation logs to understand what caused the problem and take the necessary action for the installation to resume.

    Figure 3-15: Troubleshooting (right)



OpenShift Web Console troubleshooting

In order to access the OpenShift Web Console, use external DNS server or local configuration to resolve its hostname. To do so, either:

Option 1: Add the following records to your DNS server (recommended)

```
api.ocpdeae.amd.com       A     10.216.169.217
*.apps.ocpdeae.amd.com    A     10.216.169.251
```

Option 2: Update your local /etc/hosts or /etc/resolv.conf files

20. Add the cluster web console URL DNS in order to access the console.

    - For a Linux host, this will be in `/etc/hosts`; execute the command `nano /etc/hosts` to open this file for editing, then paste the host URLs with their associated IP addresses, and then save your changes.
    - For a Windows host, open Notepad or another text editor as the Administrator, then add the URL to `C:\Windows\System32\drivers\etc\hosts`.

Your OpenShift setup is complete. You can now build and deploy applications on the installed cluster.

3.2 Installing Red Hat OpenShift AI

Red Hat OpenShift AI enables deploying LLMs using a single model serving platform based on the KServe component. This platform provides a robust and efficient hosting platform that streamlines the deployment, monitoring, scaling, and maintenance of LLMs. The single model serving platform consists of the following components:

- **KServe:** Kubernetes CRD that orchestrates model serving for all types of models. It includes model-serving runtimes that implement the loading of given types of model servers. KServe handles the life cycle of the deployment object, storage access, and networking setup.
- **Red Hat OpenShift Serverless:** Cloud-native development model that allows for serverless model deployments and is based on the open-source Knative* project.
- **The Red Hat OpenShift Service Mesh:** Networking layer that manages traffic flows and enforces access policies. It is based on the open-source Istio* project.

You may choose either an automatic or manual installation, as follows:

- **Automated:** If neither the ServiceMeshControlPlane nor KNativeServing resource is not already created on the OpenShift cluster, then configure the Red Hat OpenShift Data Science Operator to install KServe and its dependencies. See Configuring automated installation of KServe*.
- **Manual:** If either the ServiceMeshControlPlane or KNativeServing resource is already created on the OpenShift cluster, then Red Hat OpenShift Data Science Operator cannot be configured to install KServe and its dependencies and you must proceed with manual installation.

3.3 Installing KServe Dependencies

You must create Red Hat OpenShift Service Mesh and Knative Serving instances and then configure secure gateways for Knative Serving before installing KServe.

3.3.1 Step 1: Create an OpenShift Service Mesh Instance – Istio

The following prerequisites must be met in order create a Red Hat OpenShift Service Mesh instance:

- You must have cluster administrator privileges for the OpenShift Container Platform cluster.
- The cluster must have a node with 4 CPUs and 16 GB memory.
- You must have already downloaded and installed OpenShift Command Line Interface (CLI). See Installing the OpenShift CLI*.
- You must have installed the Red Hat OpenShift Service Mesh Operator* and dependent Operators.

To create an OpenShift Service Mesh instance - Istio:

1. Open a terminal window, then execute the following command to login to your OpenShift cluster CLI as a cluster administrator:

```
$ oc login <openshift_cluster_url> -u <admin_username> -p <password>
```

2. Create the required namespace for Red Hat OpenShift Service Mesh.

```
$ oc create ns istio-system
```

   The output appears as follows:

```
namespace/istio-system created
```

3. Define a ServiceMeshControlPlane object in a YAML file named `smcp.yaml` with the following contents:

```
apiVersion: maistra.io/v2
kind: ServiceMeshControlPlane
metadata:
  name: minimal
  namespace: istio-system
spec:
  tracing:
    type: None
  addons:
    grafana:
      enabled: false
    kiali:
      name: kiali
      enabled: false
    prometheus:
      enabled: false
    jaeger:
      name: jaeger
  security:
    dataPlane:
      mtls: true
    identity:
      type: ThirdParty
  techPreview:
    meshConfig:
      defaultConfig:
        terminationDrainDuration: 35s
  gateways:
    ingress:
      service:
        metadata:
          labels:
            knative: ingressgateway
  proxy:
    networking:
      trafficControl:
        inbound:
          excludedPorts:
            - 8444
            - 8022
```

4. Create the service mesh control plane.

```
$ oc apply -f smcp.yaml
```

5. In the OpenShift CLI, execute the following command to list all running pods in the `istio-system` project. This is the project where the OpenShift Service Mesh is installed:

```
 $ oc get pods -n istio-system
```

6. Confirm that there are running pods for the service mesh control plane, ingress gateway, and egress gateway. These pods have the following naming patterns:

```
NAME                                        READY     STATUS      RESTARTS      AGE
istio-egressgateway-7c46668687-fzsqj          1/1     Running            0      22h
istio-ingressgateway-77f94d8f85-fhsp9         1/1     Running            0      22h
istiod-data-science-smcp-cc8cfd9b8-2rkg4      1/1     Running            0      22h
```

3.3.2 Step 2: Create a Knative Serving Instance

The following prerequisites must be met in order to install Knative Serving and then create an instance.

- You have cluster administrator privileges for the OpenShift Container Platform cluster.
- The cluster has a node with 4 CPUs and 16 GB memory.
- You have downloaded and installed the OpenShift CLI. See Installing the OpenShift CLI*.
- You have created a Red Hat OpenShift Service Mesh instance.
- You have installed the Red Hat OpenShift Serverless Operator.

To create a Knative Serving Instance:

1. Open a terminal window, then execute the following command to login to your OpenShift cluster CLI as a cluster administrator:

```
$ oc login <openshift_cluster_url> -u <admin_username> -p <password>
```

2. Verify that the required project (that is, namespace) for Knative Serving already exists.

```
$ oc get ns knative-serving
```

   The output appears as follows:

```
NAME                       STATUS     AGE
knative-serving            Active   4d20h
```

3. Create the knative-serving project if it does not already exist:

```
$ oc create ns knative-serving
```

   The output appears as follows:

```
namespace/knative-serving created
```

4. Define a ServiceMeshMember object in a YAML file called `default-smm.yaml` with the following contents:

```
apiVersion: maistra.io/v1
kind: ServiceMeshMember
metadata:
  name: default
  namespace: knative-serving
spec:
  controlPlaneRef:
    namespace: istio-system
    name: minimal
```

5. Create the ServiceMeshMember object in the istio-system namespace.

```
$ oc apply -f default-smm.yaml
```

The output appears as follows:

```
servicemeshmember.maistra.io/default created
```

6. Define a KnativeServing object in a YAML file called `knativeserving-istio.yaml` with the following contents:

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
  annotations:
    serverless.openshift.io/default-enable-http2: "true"
spec:
  deployments:
    - annotations:
        sidecar.istio.io/inject: "true" -----(1)
        sidecar.istio.io/rewriteAppHTTPProbers: "true ----(2)
      name: activator
    - annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
      name: autoscaler
  ingress:
    istio:
      enabled: true
  config:
    features:
      kubernetes.podspec-affinity: enabled
      kubernetes.podspec-nodeselector: enabled
      kubernetes.podspec-tolerations: enabled
```

The preceding file defines a Custom Resource (CR) for a KnativeServing object. The CR also adds the following actions to each of the activator and autoscaler pods:

- Injects an Isitio sidecar to the pod. This makes the pod part of the service mesh.
- Enables the Istio sidecar to rewrite the HTTP liveness and readiness probes for the pod.

7. Create the `KnativeServing` object in the specified knative-serving namespace.

```
$ oc apply -f knativeserving-istio.yaml
```

The output appears as follows:

```
knativeserving.operator.knative.dev/knative-serving created
```

8. Review the default `ServiceMeshMemberRoll` object in the istio-system namespace.

```
$ oc describe smmr default -n istio-system
```

9. In the description of the `ServiceMeshMemberRoll` object, locate the `Status.Members` field and confirm that it includes the knative-serving namespace.

10. Verify creation of the Knative Serving instance by executing the following command in the OpenShift CLI to list all running pods in the `knative-serving` project. This is the project in which you created the Knative Serving instance.

```
$ oc get pods -n knative-serving
```

Confirm that there are numerous running pods in the `knative-serving` project, including `activator`, `autoscaler`, `controller`, and `domainmapping` pods, as well as pods for the Knative Istio controller, which controls the integration of OpenShift Serverless and OpenShift Service Mesh. The output should appear as follows:

```
NAME                                       READY     STATUS     RESTARTS      AGE
activator-7586f6f744-nvdlb                   2/2     Running           0      22h
activator-7586f6f744-sd77w                   2/2     Running           0      22h
autoscaler-764fdf5d45-pv298                  2/2     Running           0      22h
autoscaler-764fdf5d45-x7dc6                  2/2     Running           0      22h
autoscaler-hpa-7c7c4cd96d-2lkzg              1/1     Running           0      22h
autoscaler-hpa-7c7c4cd96d-gks9j              1/1     Running           0      22h
controller-5fdfc9567c-6cj9d                  1/1     Running           0      22h
controller-5fdfc9567c-bf5x7                  1/1     Running           0      22h
domain-mapping-56cc85968-2hjvp               1/1     Running           0      22h
domain-mapping-56cc85968-lg6mw               1/1     Running           0      22h
domainmapping-webhook-769b88695c-gp2hk       1/1     Running           0      22h
domainmapping-webhook-769b88695c-npn8g       1/1     Running           0      22h
net-istio-controller-7dfc6f668c-jb4xk        1/1     Running           0      22h
net-istio-controller-7dfc6f668c-jxs5p        1/1     Running           0      22h
net-istio-webhook-66d8f75d6f-bgd5r           1/1     Running           0      22h
net-istio-webhook-66d8f75d6f-hld75           1/1     Running           0      22h
webhook-7d49878bc4-8jxbr                     1/1     Running           0      22h
webhook-7d49878bc4-s4xx4                     1/1     Running           0      22h
```

3.3.3 Step 3: Create Secure Gateways for Knative Serving

You must create Secure gateways for Knative Serving instance in order to defend traffic between the Knative Serving instance and the service mesh. This procedure has the following prerequisites:

- You must have cluster administrator privileges for you OpenShift Container Platform cluster.
- You must have downloaded and installed the OpenShift CLI. See Installing the OpenShift CLI*.
- You must have created a Red Hat OpenShift Service Mesh instance.
- You must have created a Knative Serving instance.
- If you intend to generate a wildcard certificate and key, then you must download and install OpenSSL.

To create a secure gateway:

1. Open a terminal window, then execute the following command to login to your OpenShift cluster CLI as a cluster administrator:

```
$ oc login <openshift_cluster_url> -u <admin_username> -p <password>
```

2. Set environment variables to define base directories for generating a wildcard certificate and key for the gateways.

```
$ export BASE_DIR=/tmp/kserve
$ export BASE_CERT_DIR=${BASE_DIR}/certs
```

3. Set an environment variable to define the common name used by the OpenShift cluster ingress controller .

```
$ export COMMON_NAME=$(oc get ingresses.config.openshift.io cluster -o
jsonpath='{.spec.domain}' | awk -F'.' '{print $(NF-1)"."$NF}')
```

4. Set an environment variable to define the domain name used by the OpenShift cluster ingress controller.

```
$ export DOMAIN_NAME=$(oc get ingresses.config.openshift.io cluster -o
jsonpath='{.spec.domain}')
```

5. Create the required base directories for the certificate generation, based on the environment variables that were previously set.

```
$ mkdir ${BASE_DIR}
$ mkdir ${BASE_CERT_DIR}
```

6. Create the OpenSSL configuration for generating a wildcard certificate.

```
zcat <<EOF> ${BASE_DIR}/openssl-san.config
[ req ]
distinguished_name = req
[ san ]
subjectAltName = DNS:*.${DOMAIN_NAME}
EOF
```

7. Generate a root certificate.

```
$ openssl req -x509 -sha256 -nodes -days 3650 -newkey rsa:2048 \
-subj "/O=Example Inc./CN=${COMMON_NAME}" \
-keyout $BASE_DIR/root.key \
-out $BASE_DIR/root.crt
```

8. Generate a wildcard certificate signed by the root certificate.

```
$ openssl req -x509 -newkey rsa:2048 \
-sha256 -days 3560 -nodes \
-subj "/CN=${COMMON_NAME}/O=Example Inc." \
-extensions san -config ${BASE_DIR}/openssl-san.config \
-CA $BASE_DIR/root.crt \
-CAkey $BASE_DIR/root.key \
-keyout $BASE_DIR/wildcard.key  \
-out $BASE_DIR/wildcard.crt

$ openssl x509 -in ${BASE_DIR}/wildcard.crt -text
```

9. Verify the wildcard certificate.

```
$ openssl verify -CAfile ${BASE_DIR}/root.crt

{BASE_DIR}/wildcard.crt
```

10. Export the wildcard key and certificate that were created by the script to new environment variables.

```
$ export TARGET_CUSTOM_CERT=${BASE_CERT_DIR}/wildcard.crt
$ export TARGET_CUSTOM_KEY=${BASE_CERT_DIR}/wildcard.key
```

11. If desired, you may choose to export the wildcard key and certificate to new environment variables by executing the following commands:

```
$ export TARGET_CUSTOM_CERT=<path_to_certificate>
$ export TARGET_CUSTOM_KEY=<path_to_key>
```

12. You must specify the domain name used by the OpenShift cluster ingress controller in the certificate that was provided. Execute the following command to check this value:

```
$ oc get ingresses.config.openshift.io cluster -o jsonpath='{.spec.domain}'
```

13. Create a TLS secret in the istio-system namespace using the environment variables that were set for the wildcard certificate and key.

```
$ oc create secret tls wildcard-certs --cert=${TARGET_CUSTOM_CERT} --
key=${TARGET_CUSTOM_KEY} -n istio-system
```

14. Create a `gateways.yaml` YAML file with the following contents:

```
apiVersion: v1
kind: Service -----(1)
metadata:
  labels:
    experimental.istio.io/disable-gateway-port-translation: "true"
  name: knative-local-gateway
  namespace: istio-system
spec:
  ports:
    - name: http2
      port: 80
      protocol: TCP
      targetPort: 8081
  selector:
    knative: ingressgateway
  type: ClusterIP
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: knative-ingress-gateway -------(2)
  namespace: knative-serving
spec:
  selector:
    knative: ingressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: https
        number: 443
        protocol: HTTPS
      tls:
        credentialName: wildcard-certs
        mode: SIMPLE
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
name: knative-local-gateway -------(3)
namespace: knative-serving
spec:
selector:
   knative: ingressgateway
servers:
   - port:
       number: 8081
       name: https
       protocol: HTTPS
     tls:
       mode: ISTIO_MUTUAL
     hosts:
       - "*"
```

This YAML file:

- Defines a service in the istio-system namespace for the Knative local gateway.
- Defines an ingress gateway in the knative-serving namespace. The gateway uses the TLS secret you created earlier in this procedure. The ingress gateway handles external traffic to Knative.
- Defines a local gateway for Knative in the knative-serving namespace.

15. Apply the `gateways.yaml` file to create the defined resources.

```
$ oc apply -f gateways.yaml
```

The output appears as follows:

```
service/knative-local-gateway created
gateway.networking.istio.io/knative-ingress-gateway created
gateway.networking.istio.io/knative-local-gateway created
```

16. Review the gateways that were created, being sure to verify that the local and ingress gateways were created in the knative-serving namespace.

```
$ oc get gateway --all-namespaces
```

The output appears as follows:

```
NAMESPACE          NAME                                        AGE
knative-serving    knative-ingress-gateway                     69s
knative-serving    knative-local-gateway                        2m
```

3.3.4 Step 4: Install KServe

You must install the Red Hat OpenShift Data Science Operator in order to complete manual KServe installation. You can configure the Operator to install KServe. This requires meeting the following prerequisites:

- You must have cluster administrator privileges for OpenShift Container Platform cluster.
- The cluster must have a node with 4 CPUs and 16 GB memory.
- You must have downloaded and installed the OpenShift CLI. See Installing the OpenShift CLI*.
- You must have created a Red Hat OpenShift Service Mesh instance, a Knative Serving instance, and secure gateways for Knative Serving.
- You must have installed the Red Hat OpenShift Data Science Operator and created a DataScienceCluster object.

To install KServe:

1. Login to the OpenShift web console as a cluster administrator.
2. In the web console, select **Operators>Installed Operators**, and then click **Red Hat OpenShift Data Science Operator**.
3. Configure the OpenShift Service Mesh component as follows:
   a. Access the **DSC Initialization** tab.
   b. Select the **default-dsci** object.
   c. Access the **YAML** tab.
   d. In the **spec** section, add and configure the `serviceMesh` component as shown:

```
spec:
serviceMesh:
   managementState: Unmanaged
```

4. Click **Save**.

5. Configure the KServe and OpenShift Serverless components as follows:

    a. In the web console, select **Operators>Installed Operators**, and then select **Red Hat OpenShift Data Science Operator**.

    b. Access the **Data Science Cluster** tab.

    c. Select the `default-dsc` DSC object.

    d. Select the **YAML** tab.

    e. In the `spec.components` section, configure the `kserve` component as shown:

```
spec:
components:
   kserve:
     managementState: Managed
```

6. Within the kserve component, add and configure the serving component as shown:

```
spec:
components:
   kserve:
     managementState: Managed
     serving:
       managementState: Unmanaged
```

7. Click **Save**.

Chapter 4 Using TGI to Deploy the Llama2 Model on a Single Docker Container

4.1 Obtaining a Llama2 Model

To get a LLama2 model:

1. Create a Hugging Face* account.
2. Request access to gated models.
3. At Hugging Face, navigate to meta/LLama2, then request access to LLama2 models.
4. Once approval is granted, create a Hugging Face access token*. You will use this token to download models and deployments.

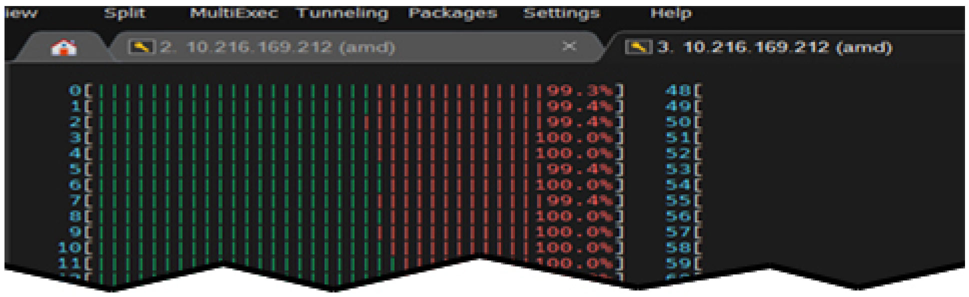
4.2 CPU Optimization When Running Llama2 Model as a Single Docker Container

You can optimize processor performance by pinning vCPUs to the Docker container that runs the TGI Inference server. CPU pinning (or processor affinity) attaches specified cores with `core_ids` to the specific processor (which is a container in this example). This also reduces latency by avoiding cross-socket communication.

The `cpuset-cpus` argument sets the CPU pinning for a container. For example:

```
$ docker run --cpus 48 --cpuset-cpus "0-47" --shm-size 1g -e HUGGING_FACE_HUB_TOKEN=$token
-p 8080:80 -v $volume:/data ghcr.io/huggingface/text-generation-inference:1.3 --model-id
$model
```

The output will appear as follows:



```
  0[  99.3%]   48[
  1[  99.4%]   49[
  2[  99.4%]   50[
  3[ 100.0%]   51[
  4[ 100.0%]   52[
  5[  99.4%]   53[
  6[ 100.0%]   54[
  7[  99.4%]   55[
  8[ 100.0%]   56[
  9[ 100.0%]   57[
 10[ 100.0%]   58[
 11[ 100.0%]   59[
```

Figure 4-1: Container CPU pinning

4.3 Deploying a Llama2-7B Model with TGI on a Single Docker Container

TGI can serve a LLM as a Docker container. Here is a simple Llama-2-7b model deployment example:

1. Set the following required parameters:

```
model=meta-llama/Llama-2-7b-chat-hf
volume=$PWD/data # share a volume with the Docker container to avoid downloading weights
every
token=<your cli READ token>
```

2. Execute the following command to deploy a TGI inference for `Llama-2-7b-chat-hf`:

```
$ docker run –cpus all --shm-size 1g -e HUGGING_FACE_HUB_TOKEN=$token -p 8080:80 -v
$volume:/data ghcr.io/huggingface/text-generation-inference:1.3 --model-id $model
```

This example deploys and serves the LLM at port 8080. The sample LLM endpoint is therefore http//:localhost:8080/generate_stream.

Chapter 5 CPU Optimization for Serving Models on Red Hat OCP

You may optionally enable multiple optimizations to boost OpenShift throughput while lower latency, such as:

- **CPU Manager Policy:** See "Enabling the CPU-Manager Policy" on page 31.
- **Topology Manager:** See "Setting the Topology Manager Policy" on page 34.
- **Feature gate:** Useful for tech previews. This method is beyond the scope of this User Guide.

5.1 Enabling the CPU-Manager Policy

CPU Manager manages groups of CPUs and constrains workloads to specific CPUs. CPU manager policies control how CPU resources are allocated to containers within pods and offer a way to fine-tune performance and isolate resources for a specific workload.

The `static` policy allocates CPU resources across different types of pods as follows:

- Guaranteed pods receive the highest priority for CPU resources. They specify an integer number of CPU requests and have exclusive access to those CPUs for the entire duration of their existence. This ensures consistent performance for critical workloads.
- Best-Effort pods receive CPU resources only when no Guaranteed or Burstable pods are using them. They have no guaranteed CPU allocation and experience variable performance depending on available resources. Use them for non-critical workloads that can tolerate fluctuations.
- Burstable pods fall between Guaranteed and Best-Effort pods. They request a specific amount of CPU and have a burst limit defining additional CPU they can use for short periods. Burstable pods that exceed their base request compete with Best-Effort pods for the remaining resources, which may impact their performance.

The CPU manager periodically writes resource updates through the CRI in order to reconcile in-memory CPU assignments with `cgroupfs`. The Kubelet `--cpu-manager-reconcile-period` configuration value sets the reconcile frequency.

To enable this policy:

1. If desired, you may label a node.

```
$ oc label node perf-node.example.com cpumanager=true
```

2. Edit the `MachineConfigpool` of the nodes where the CPU Manager should be enabled.

```
$ oc edit machineconfigpool worker
```

3. Add a label to the worker machine configuration pool:

```
metadata:
  creationTimestamp: 2020-xx-xxx
  generation: 3
  labels:
    custom-kubelet: cpumanager-enabled
```

4. Create `KubeletConfig, cpumanager-kubeletconfig.yaml` with the following contents:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
     cpuManagerPolicy: static  ------(1)
     cpuManagerReconcilePeriod: 5s -------(2)
```

In this YAML file:

- `cpuManagerPolicy` specifies the CPU Manager policy to use, such as `static`.
- `cpuManagerReconcilePeriod` sets the reconciliation frequency. The default is `5s`. This value is optional.

5. Create a dynamic kubelet config:

```
$ oc create -f cpumanager-kubeletconfig.yaml
```

This adds the CPU Manager feature to the kubelet config. If needed, the Machine Config Operator (MCO) reboots the node. A reboot is not needed to enable CPU Manager.

6. Check for the merged kubelet config:

```
$ oc get machineconfig 99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet -o json | grep
ownerReference -A7
```

The output appears as follows:

```
     "ownerReferences": [
            {
                "apiVersion": "machineconfiguration.openshift.io/v1",
                "kind": "KubeletConfig",
                "name": "cpumanager-enabled",
                "uid": "7ed5616d-6b72-11e9-aae1-021e1ce18878"
            }
        ]
```

7. Check the worker for the updated `kubelet.conf`:

```
$ oc debug node/<node-name>
$ sh-4.2# cat /host/etc/kubernetes/kubelet.conf | grep cpuManager
```

The output appears as follows:

```
cpuManagerPolicy: static
cpuManagerReconcilePeriod: 5s
```

8. Create a pod and deploy it on the OpenShift cluster using guaranteed resources after enabling the `cpuManager` policy.

9. Describe the pod by executing the following command for verification.

```
$ oc describe pod <pod-name>
```

The output appears as follows:

```
Name:                 cpumanager-6cqz7
Namespace:            default
Priority:             0
PriorityClassName:    <none>
Node:  perf-node.example.com/xxx.xx.xx.xxx
...
Limits:
      cpu:     1
      memory:  1G
    Requests:
      cpu:        1
      memory:     1G
...
QoS Class:       Guaranteed
Node-Selectors:  cpumanager=true
```

10. Verify that the `cgroups` are set up correctly. Get the process ID (PID) of the pause process:

```
$ systemd-cgls
```

The output appears as follows:

```
├─init.scope
│ └─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 17
└─kubepods.slice
  ├─kubepods-pod69c01f8e_6b74_11e9_ac0f_0a2b62178a22.slice
  │ ├─crio-b5437308f1a574c542bdf08563b865c0345c8f8c0b0a655612c.scope
  │ └─32706 /pause
```

11. Place pods of Quality of Service (QoS) tier Guaranteed within the `kubepods.slice`. Place pods of other QoS tiers in child `cgroups` of `kubepods`:

```
$ cd /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-
pod69c01f8e_6b74_11e9_ac0f_0a2b62178a22.slice/crio-
b5437308f1ad1a7db0574c542bdf08563b865c0345c86e9585f8c0b0a655612c.scope
$ for i in `ls cpuset.cpus tasks` ; do echo -n "$i "; cat $i ; done
```

The output appears as follows:

```
cpuset.cpus 1
tasks 32706
```

12. Check the allowed CPU list for the task:

```
$ grep ^Cpus_allowed_list /proc/32706/status
```

The output appears as follows:

```
$ Cpus_allowed_list: 1
```

5.2 Setting the Topology Manager Policy

The Topology Manager is a crucial Kubelet component that provides an interface for components (known as Hint Providers) to send and receive topology information. It ensures that resource allocation decisions consider the underlying hardware topology, such as NUMA nodes. For example, it collects hints from the CPU Manager, Device Manager, and other Hint Providers to align pod resources (e.g., CPU, SR-IOV VFs, and other device resources) for QoS tiers on the same Non-Uniform Memory Access (NUMA) node. Topology Manager is useful for workloads that use hardware accelerators to support latency-critical execution and high-throughput parallel computation. The following prerequisites must be met:

- Ensure that the CPU Manager is enabled (`cpumanager-enabled`).
- Set the CPU Manager policy to `static`.

To set the Topology Manager police:

1. Configure the Topology Manager policy in the `cpumanager-enabled` CR.

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
     cpuManagerPolicy: static
     cpuManagerReconcilePeriod: 5s
     topologyManagerPolicy: single-numa-node -----(1)
```

2. Specify the selected Topology Manager policy (`single-numa-node` in this example). Acceptable values are: `default`, `best-effort`, `restricted`, or `single-numa-node`.
3. Check if the Topology Manager policy is applied:

```
$ oc get nodes
```

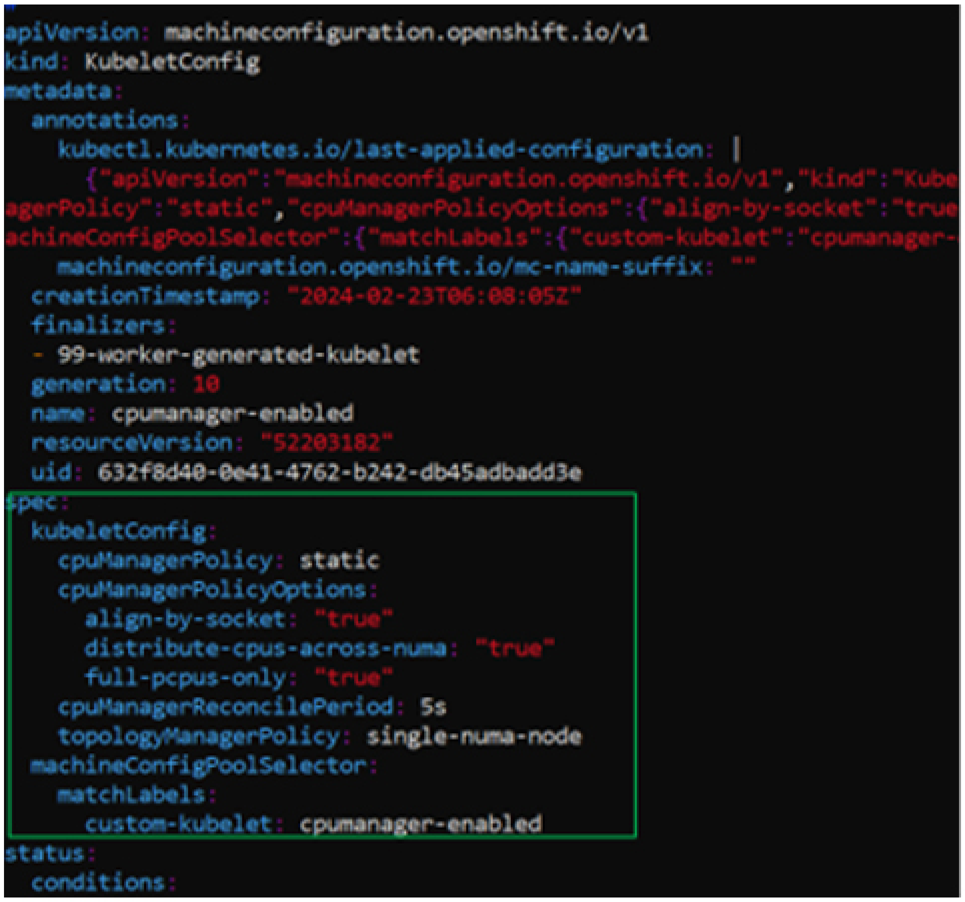
4. After getting all the nodes, get inside any worker node:

```
$ oc debug node/aifinal-worker3
```

5. Print the `kubelet.conf` file and ensure the changes are applied:

```
$ cat /etc/kubernetes/kubelet.conf
```

The output will appear as follows:



```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"machineconfiguration.openshift.io/v1","kind":"Kube
agerPolicy":"static","cpuManagerPolicyOptions":{"align-by-socket":"true
achineConfigPoolSelector":{"matchLabels":{"custom-kubelet":"cpumanager-
    machineconfiguration.openshift.io/mc-name-suffix: ""
  creationTimestamp: "2024-02-23T06:08:05Z"
  finalizers:
  - 99-worker-generated-kubelet
  generation: 10
  name: cpumanager-enabled
  resourceVersion: "52203182"
  uid: 632f8d40-0e41-4762-b242-db45adbadd3e
spec:
  kubeletConfig:
    cpuManagerPolicy: static
    cpuManagerPolicyOptions:
      align-by-socket: "true"
      distribute-cpus-across-numa: "true"
      full-pcpus-only: "true"
    cpuManagerReconcilePeriod: 5s
    topologyManagerPolicy: single-numa-node
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
status:
  conditions:
```

Figure 5-1: kublet.conf file with changes applied

This page intentionally left blank.

Chapter 6

Deploying the Llama2 Model Using Kserve & TGI on Red Hat OCP

Red Hat OpenShift AI includes a model-serving stack that is based on the Compositional AI Toolkit (Caikit), Text Generation Inference Service (TGIS), and KServe components for serving Large Language Models. The stack helps to deploy, monitor, scale, and maintain LLMs.

6.1 User Interface Deployment

Deploying Llama2 via the user interface requires meeting the following prerequisites:

- You must haver cluster administrator privileges for the OCP cluster.
- The cluster has a node with 4 CPUs and 16 GB memory.
- You have downloaded and installed the OpenShift CLI. See Installing the OpenShift CLI*.
- You have installed the Red Hat OpenShift Service Mesh Operator* and dependent Operators.

6.2 Deploying Llama2 on Red Hat OpenShift AI

6.2.1 Step 1: Setup MinIO

Setup MinIO (access the dashboard) and create a bucket, Keep the following details: `access_key`, `secret_key`, `api_endpoint`, and `bucket_name`).

1. Execute the following commands to download and install the latest stable MinIO DEB:

```
$ wgethttps://dl.min.io/server/minio/release/linuxamd64/archive/
minio_20240118225128.0.0_amd64.deb -O minio.deb

$ sudodpkg -iminio.deb
```

2. Execute the following command from the system terminal or shell to start a local MinIO instance using the `~/minio` folder. You can replace this path with another folder path on the local machine:

```
$ mkdir ~/minio
$ minio server ~/minio --console-address :9001
```

The output appears as follows:

```
API: http://192.0.2.10:9000  http:..127.0.01.:9000
RootUser: minioadmin
```

```
RootPass: minioadmin

Console: http://192.0.2.10:9001  http:..127.0.01.:9001
RootUser: minioadmin
RootPass: minioadmin

Command-line: https://min.io/docs/minio/linux/reference/minio-mc
   $ mc alias set myminio http://192.0.2.10:9000 minioadmin minioadmin

Documentation: https://min.io/docs/minio/linux/index.html

WARNING: Detected default credentials 'minioadmin:minioadmin', w
```

3. Connect the browser to the MinIO server.
   - Open a web browser and navigate to `http://127.0.0.1:9001` to access the MinIO Console.
   - Login to the Console with the root username and password displayed in the output. The default is `minioadmin:minioadmin`.

6.2.2 Step 2: Convert the Model

Converting the desired model into caikit format and upload the model into the bucket created in MinIO.

1. Create a virtual environment with Python 3.9 and install `caikit-nlp`:

```
$ python3.9 -m venvvenv
$ Source ./venv/bin/activate
$ pip install caikit-nlp
```

2. If the model has not already been downloaded, then setting the environment variable `ALLOW_DOWNLOADS=1` will allow direct downloads of models from the Hugging Face Hub:

```
$ export ALLOW_DOWNLOADS=1
```

3. Execute the following python snippet.

```
import caikit_nlp
model_name="MODELNAME" # modify as required
converted_model_path=f"{model_name}-caikit"
caikit_nlp.text_generation.TextGeneration.bootstrap(model_name).save(converted_model_path)
print(f"Model saved to {converted_model_path}")
```

Note: If the model was already downloaded locally, then you may simply change `model_name` *to point to the path to the model.*

6.2.3 Step 3: Upload the Model

Execute the following python script to upload the model into Model Bucket:

- Set the MinIO access key, secret key, endpoint, and bucket name.
- Set the local folder path and `minio_object_prefix`.

```
import os
import boto3
from botocore.exceptions import NoCredentialsError
# Set your MinIO credentials and endpoint#
minio_access_key = "minioadmin"
minio_secret_key = "minioadmin"
minio_endpoint = "http://10.216.173.149:9000"  # Update with your MinIO server's endpoint
minio_bucket_name = "data"
# Set the local folder path and object prefix (the prefix under which the folder contents 
will be stored in MinIO)#
local_folder_path = "/mnt/models/meta-llama/Llama-2-13b-chat-hf-caikit"
minio_object_prefix = "Llama-2-13b-chat-hf-caikit"
# Update with your desired object prefix

def upload_folder_to_minio(local_path, bucket_name, object_prefix, access_key, secret_key, 
endpoint_url):
    # Create an S3 client
    s3 = boto3.client("s3", aws_access_key_id=access_key, 
aws_secret_access_key=secret_key, endpoint_url=endpoint_url)

    for root, dirs, files in os.walk(local_path):
        for filename in files:
local_file_path = os.path.join(root, filename)
relative_path = os.path.relpath(local_file_path, local_path)
minio_object_key = os.path.join(object_prefix, relative_path)

            try:
                # Upload each file
                s3.upload_file(local_file_path, bucket_name, minio_object_key)
print(f"File '{local_file_path}' uploaded successfully to '{bucket_name}/
{minio_object_key}'")
            except NoCredentialsError:
print("Credentials not available or incorrect.")


# Upload the folder to MinIO
upload_folder_to_minio(local_folder_path, minio_bucket_name, minio_object_prefix, 
minio_access_key, minio_secret_key, minio_endpoint)
```

6.2.4 Serve the Model

1. In the left menu, select **Data Science Projects**.
2. Click the project name where you want to deploy a model.
3. Execute one of the following action in the in the **Models** and **Model Servers** sections:
   - If you are using a single model serving platform tile, then click **Deploy model** on the tile.
   - Otherwise, click the **Deploy model** button.

   The **Deploy model** dialog opens.

4. Configure the following model deployment properties:
   - **Model name:** Enter a unique name for the model being deployed.
   - **Serving runtime:** Select **Caikit TGIS ServingRuntime for KServe**.
   - **Model framework:** Select **caikit**.
   - **Number of model replicas to deploy:** Specify a value.
   - **Model server size:** Select a value.
5. Specify the model location by performing either of the following procedures:

   To use an existing data connection

   a. Select **Existing data connection**.
   b. In the **Name** list, select a previously-defined data connection.
   c. In the **Folder path** field, enter the folder path that contains the model in the specified data source.

   To use a new data connection

   a. **Name:** Enter a unique name for the data connection.
   b. **Access key:** Enter the access key ID for the S3-compatible object storage provider.
   c. **Secret key:** Enter the secret access key for the specified S3-compatible object storage account.
   d. **Endpoint:** Enter the endpoint of the S3-compatible object storage bucket.
   e. **Region:** Enter the default region of the S3-compatible object storage account.
   f. **Bucket:** Enter the name of the S3-compatible object storage bucket.
   g. **Folder path:** Enter the folder path in the S3-compatible object storage that contains the data file.
   h. Click **Deploy**.

You can now access the LLM endpoint from the dashboard. The sample endpoint is `https://llama2-7b-48-predictor-new-llama2.apps.ai-final.amd.com:443/api/v1/task/server-streaming-text-generation`.

```
$ https://<deploymentname><namespace><domainname>:443/api/v1/task/server-streaming-text-
generation.
```

6.3 TGI Inference Server Deployment Without Auto-Scaling (YAML Based Deployment)

Deploying Llama2-7b on an OpenShift cluster is done using a Deployment YAML file using the Text-Generation Image, and then adding the required configurations. Apply the file to the cluster, then expose the service is for the deployment, and then create the route. The endpoint is then used to send the request using the `curl` command.

6.3.1 Rebuilding the TGI Docker Image for Deploying Llama2 7B Model on OCP

Use port 8080 instead of the default OpenShift cluster binding port (80) that is exposed for TGI Inference server.

1. Clone the TGI github repository*.
2. Change the default port number in the TGI dockerfile to 8080.
3. Build the Docker Image.

```
$ docker build -t <my_image>:latest .
```

4. Tag the Image:

```
$ docker tag my_image:latest my_registry/my_image:latest
```

5. Push the image to the repository:

```
$ docker push my_registry/my_image:latest
```

6.3.2 Deployment

1. Create a YAML file called `deploymentgi.yaml`.
   - Add the TGI model Serving Image name to the YAML file.
   - Provide the `model-id` and `HUGGINGFACE_TOKEN` variables.
   - Add all required configurations.

```
##deploymentgi.yaml##

apiVersion: apps/v1
kind: Deployment
metadata:
  name: tgi-inference-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tgi-inference
  template:
    metadata:
      labels:
        app: tgi-inference
    spec:
      containers:
        - name: tgi-inference-container
          image: akhil2507/tgi_images:v1
          ports:
            - containerPort: 8080
          args:
            - "--model-id"
            - "meta-llama/Llama-2-7b-chat-hf"  # Replace with your actual model ID
            - "--dtype"
            - "bfloat16"
          env:
            - name: HUGGING_FACE_HUB_TOKEN
              value: <HUGGING_FACE_TOKEN>
# Replace with your actual Hugging Face token
          resources:
            requests:
              memory: "48Gi"  # Adjust as needed
              cpu: 48     # Adjust as needed
            limits:
              memory: "48Gi"   # Adjust as needed
              cpu: 48    # Adjust as needed
          volumeMounts:                    # Add this section
            - name: data-volume
              mountPath: /data
      volumes:                           # Add this section
        - name: data-volume
          emptyDir: {}
```

2. Apply `deploymentgi.yaml`.

```
$ oc apply -f deploymentgi.yaml
```

6.3.2.1 Expose Service

Retrieving the endpoint requires exposing the service and creating routes. To expose the service:

1. Retrieve the deployment name.

```
$ oc get deployments
```

2. Execute the following command with the deployment name to expose the service:

```
$ oc expose deployment <deployment-name>
```

6.3.2.2 Create Routes

Create routes after exposing the service.

1. Access the OpenShift platform user interface, then navigate to the **Administration** dashboard.
2. In the left-hand menu, select **Networking** > **Routes** > **Create routes**.
3. Select **Configure via form view**.
4. Add the name for the route related to the deployment.
5. Select the **Service** and **Target port**.

6. Click **Create**.

6.3.3 LLM API Endpoint

You can now access the endpoint in the dashboard in the location column corresponding to the route name and can also send requests to the endpoint. The sample endpoint is http://amd-testing-tgi-llama2-amd-demo.apps.ai-final.amd.com/generate_stream.

```
$ http://<routename><namespace><domainname>/generate_stream
```

6.4 TGI Inference Server Deployment with Auto-Scaling (YAML Based Deployment)

Deploy the inference service by creating a YAML file and injecting the Istio sidecars. Taking advantage of all of Istio features requires the pods in the mesh to run an Istio sidecar proxy. Istio sidecars are injected to the Inference YAML in the `annotations` section, which in turn adds the proxy configuration.

```
##inferencetgi.yaml##

apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: tgi-inference-service-web
  annotations:
    openshift.io/display-name: Llama-2-7b-chat-hf-bf16
    serving.knative.openshift.io/enablePassthrough: 'true'
    sidecar.istio.io/inject: 'true'
    sidecar.istio.io/rewriteAppHTTPProbers: 'true'
spec:
  predictor:
    containers:
      - name: predictor-container
        image: akhil2507/tgi_images:v1
        env:
          - name: "HUGGING_FACE_HUB_TOKEN"
            value: <HUGGINGFACE_TOKEN>
          - name: 'RUST_BACKTRACE'
            value: '1'
        args:
          - "--model-id"
          - "meta-llama/Llama-2-7b-chat-hf"
        ports:
          - containerPort: 8080
        resources:
          requests:
            memory: "48Gi"
            cpu: "48000m"
          limits:
            memory: "48Gi"
            cpu: "48000m"
        volumeMounts:
          - name: data-volume
            mountPath: /data
    volumes:
      - name: data-volume
        emptyDir: {}

The YAML file (Inferencetgi.yaml) file can then be applied using the below command:
$ oc apply -f inferencetgi.yaml
```

Once the pods are running, navigate to **Serverless > Serving**. You can access the endpoint in the **Services** column and send requests to it. The sample endpoint is following https://tgi-inference-service-web-predictor-test-llama.apps.ai-final.amd.com/.

```
$ https: //<inferenceservice name><namespace><domain-name>

$ curl <endpoint> /generate \
    -X POST \
    -d '{"inputs":"What is Deep Learning?","parameters":{"max_new_tokens":20}}' \
    -H 'Content-Type: application/json'
```

6.5 Send Queries Using Curl/Request Python Scripts

You can send requests via `curl` commands once the deployment service and routes are created.

```
$ curl <endpoint> \
    -X POST \
    -d '{"inputs":"What is Deep Learning?","parameters":{"max_new_tokens":20}}' \
    -H 'Content-Type: application/json'
```

You can also run queries using the `Request.py` file.

```
#Request.py#

import requests

url = "<endpoint>"

headers = {
    "Content-Type": "application/json",
}

data = {
    "model_id": "llama2-7b-chat-hf",
    "inputs": "what is deep learning?"
}

for i in range(100):  # Send 100 requests
    try:
        response = requests.post(url, headers=headers, json=data, verify=False)
        response.raise_for_status()  # Raise an exception for non-200 status codes

        print(response.json())  # Print the response data (assuming JSON)
    except requests.exceptions.RequestException as err:
        print("Request failed:", err)
```

6.6 Verifying Replicas Serving Model Pinned to a Single Socket

Execute the following commands to scale out replicas for the any deployments:

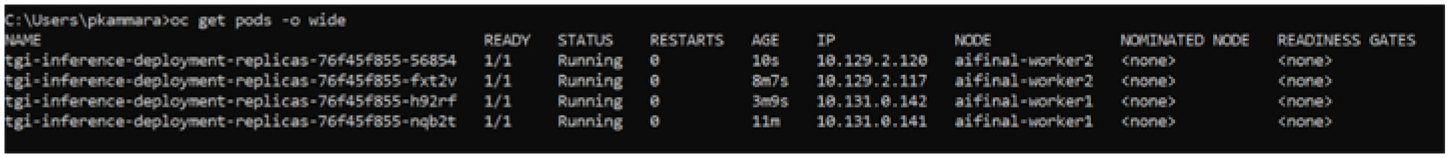
1. To get the deployment name:

```
$ oc get deployments
```

2. To scale the replicas:

```
$ oc scale --replicas=4 deployment/<deployment-name>
```

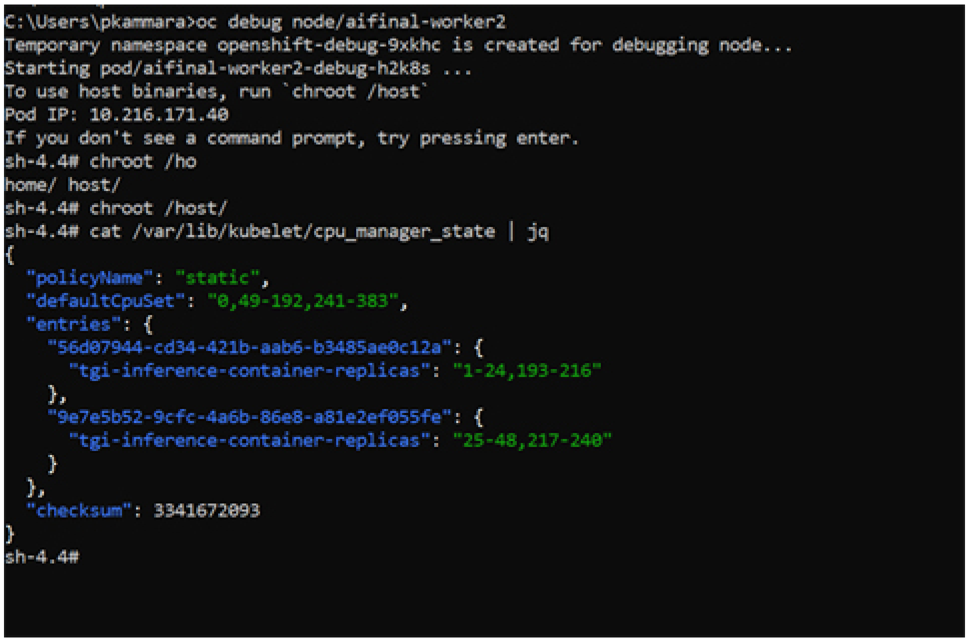
Figure 6-1 shows four replicas.



```
C:\Users\pkammara>oc get pods -o wide
NAME                                          READY   STATUS    RESTARTS   AGE    IP             NODE              NOMINATED NODE   READINESS GATES
tgi-inference-deployment-replicas-76f45f855-56854   1/1     Running   0          10s    10.129.2.120   aifinal-worker2   <none>           <none>
tgi-inference-deployment-replicas-76f45f855-fxt2v   1/1     Running   0          8m7s   10.129.2.117   aifinal-worker2   <none>           <none>
tgi-inference-deployment-replicas-76f45f855-h92rf   1/1     Running   0          3m9s   10.131.0.142   aifinal-worker1   <none>           <none>
tgi-inference-deployment-replicas-76f45f855-nqb2t   1/1     Running   0          11m    10.131.0.141   aifinal-worker1   <none>           <none>
```

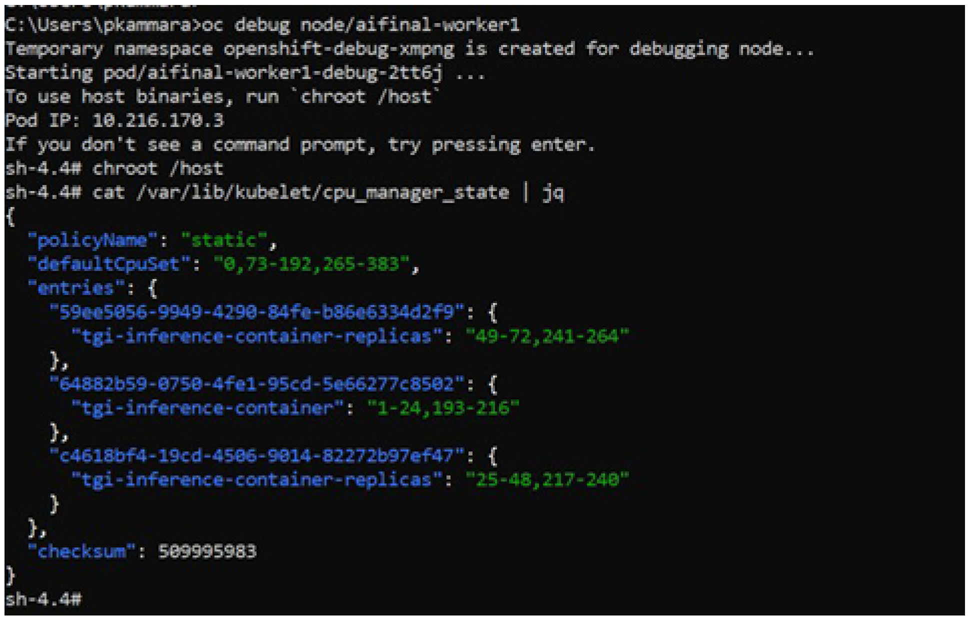
Figure 6-1: Four replicas

You can verify CPU pinning across the nodes.



```
C:\Users\pkammara>oc debug node/aifinal-worker2
Temporary namespace openshift-debug-9xkhc is created for debugging node...
Starting pod/aifinal-worker2-debug-h2k8s ...
To use host binaries, run `chroot /host`
Pod IP: 10.216.171.40
If you don't see a command prompt, try pressing enter.
sh-4.4# chroot /ho
home/ host/
sh-4.4# chroot /host/
sh-4.4# cat /var/lib/kubelet/cpu_manager_state | jq
{
  "policyName": "static",
  "defaultCpuSet": "0,49-192,241-383",
  "entries": {
    "56d07944-cd34-421b-aab6-b3485ae0c12a": {
      "tgi-inference-container-replicas": "1-24,193-216"
    },
    "9e7e5b52-9cfc-4a6b-86e8-a81e2ef055fe": {
      "tgi-inference-container-replicas": "25-48,217-240"
    }
  },
  "checksum": 3341672093
}
sh-4.4#
```

Figure 6-2: Socket pinning at Worker 2



```
C:\Users\pkammara>oc debug node/aifinal-worker1
Temporary namespace openshift-debug-xmpng is created for debugging node...
Starting pod/aifinal-worker1-debug-2tt6j ...
To use host binaries, run `chroot /host`
Pod IP: 10.216.170.3
If you don't see a command prompt, try pressing enter.
sh-4.4# chroot /host
sh-4.4# cat /var/lib/kubelet/cpu_manager_state | jq
{
  "policyName": "static",
  "defaultCpuSet": "0,73-192,265-383",
  "entries": {
    "59ee5056-9949-4290-84fe-b86e6334d2f9": {
      "tgi-inference-container-replicas": "49-72,241-264"
    },
    "64882b59-0750-4fe1-95cd-5e66277c8502": {
      "tgi-inference-container": "1-24,193-216"
    },
    "c4618bf4-19cd-4506-9014-82272b97ef47": {
      "tgi-inference-container-replicas": "25-48,217-240"
    }
  },
  "checksum": 509995983
}
sh-4.4#
```

Figure 6-3: Socket pinning at Worker 1

Chapter 7 Deploying Llama2 13B and 70B Models Using TGI on Red Hat OCP

7.1 Llama2 13B TGI Inference Server Deployment without Auto-Scaling (YAML Based Deployment)

7.1.1 Rebuilding the TGI Docker Image for Deploying Llama2 7B on and OpenShift Cluster

Use port 8080 instead of the default OpenShift cluster binding port (80) that is exposed for TGI Inference server.

1. Clone the TGI github repository*.
2. Change the default port number in the TGI dockerfile to 8080.
3. Build the Docker Image.

```
$ docker build -t <my_image>:latest .
```

4. Tag the Image:

```
$ docker tag my_image:latest my_registry/my_image:latest
```

5. Push the image to the repository:

```
$ docker push my_registry/my_image:latest
```

7.1.2 Deployment

1. Create a YAML file called `deploymentgi.yaml`.
   - Add the TGI model Serving Image name to the YAML file.
   - Provide the `model-id` and `HUGGINGFACE_TOKEN` variables.
   - Add all required configurations.

```
##deploymentgi-13b.yaml##

apiVersion: apps/v1
kind: Deployment
metadata:
  name: tgi-inference-deployment-13b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tgi-inference
  template:
    metadata:
      labels:
        app: tgi-inference
    spec:
      containers:
        - name: tgi-llama2-13b-deployment-container
          image: akhil2507/tgi_images:v1
          ports:
            - containerPort: 8080
          args:
            - "--model-id"
            - "meta-llama/Llama-2-13b-chat-hf"
            - "--dtype"
            - "bfloat16"
          env:
           - name: HUGGING_FACE_HUB_TOKEN
            value: <HUGGING FACE TOKEN>
          resources:
            requests:
              memory: "96Gi"  # Adjust as needed
              cpu: 96     # Adjust as needed
            limits:
              memory: "96Gi"   # Adjust as needed
              cpu: 96    # Adjust as needed
          volumeMounts:                    # Add this section
            - name: data-volume
              mountPath: /data
      volumes:                             # Add this section
        - name: data-volume
          persistentVolumeClaim:
            claimName: llama2-13b-claim
```

2. Apply `deploymentgi.yaml`.

```
$ oc apply -f deploymentgi.yaml
```

7.1.2.1 Expose Service

Retrieving the endpoint requires exposing the service and creating routes. To expose the service:

1. Retrieve the deployment name.

```
$ oc get deployments
```

2. Execute the following command with the deployment name to expose the service:

```
$ oc expose deployment <deployment-name>
```

7.1.2.2 Create Routes

Create routes after exposing the service.

1. Access the OpenShift platform user interface, then navigate to the **Administration** dashboard.

2. In the left-hand menu, select **Networking** > **Routes** > **Create routes**.
3. Select **Configure via form view**.
4. Add the name for the route related to the deployment.
5. Select the **Service** and **Target port**.
6. Click **Create**.

7.1.3 LLM API Endpoint

You can now access the endpoint in the dashboard in the location column corresponding to the route name and can also send requests to the endpoint. The sample endpoint is http://tgi-inference-13b-deployment-llama2-13b.apps.ai-in-a-box.amd.com/generate_stream.

```
$ http://<routename><namespace><domainname>/generate_stream
```

7.1.4 Send Queries Using Curl/Request Python Scripts

You can send requests via `curl` commands once the deployment service and routes are created.

```
$ curl <endpoint> \
    -X POST \
    -d '{"inputs":"What is Deep Learning?","parameters":{"max_new_tokens":20}}' \
    -H 'Content-Type: application/json'
```

You can also run queries using the `Request.py` file.

```
#Request.py#

import requests

url = "<endpoint>"

headers = {
    "Content-Type": "application/json",
}

data = {
    "model_id": "llama2-13b-chat-hf",
    "inputs": "what is deep learning?"
}

for i in range(100):  # Send 100 requests
    try:
        response = requests.post(url, headers=headers, json=data, verify=False)
        response.raise_for_status()  # Raise an exception for non-200 status codes

        print(response.json())  # Print the response data (assuming JSON)
    except requests.exceptions.RequestException as err:
        print("Request failed:", err)
```

7.2 Llama2 70B TGI Inference Server Deployment with Auto-Scaling (YAML Based Deployment)

Follow the process described in "Llama2 13B TGI Inference Server Deployment without Auto-Scaling (YAML Based Deployment)" on page 47, using the following `deploymentgi.yaml`:

```
##deploymentgi-70b.yaml##

apiVersion: apps/v1
kind: Deployment
metadata:
  name: tgi-inference-deployment-70b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tgi-inference
  template:
    metadata:
      labels:
        app: tgi-inference
    spec:
      containers:
        - name: tgi-llama2-70b-deployment-container
          image: akhil2507/tgi_images:v1
          ports:
            - containerPort: 8080
          args:
            - "--model-id"
            - "meta-llama/Llama-2-70b-chat-hf"
            - "--dtype"
            - "bfloat16"
          env:
           - name: HUGGING_FACE_HUB_TOKEN
            value: <HUGGING_FACE_TOKEN>
          resources:
            requests:
              memory: "512Gi" # Adjust as needed
              cpu: 192    # Adjust as needed
            limits:
              memory: "512Gi"   # Adjust as needed
              cpu: 192    # Adjust as needed
          volumeMounts:                        # Add this section
            - name: data-volume
              mountPath: /data
      volumes:                               # Add this section
        - name: data-volume
          persistentVolumeClaim:
            claimName: llama2-70b-claim
```

7.2.1 Send Queries Using Curl/Request Python Scripts

You can send requests via `curl` commands once the deployment service and routes are created.

```
$ curl <endpoint> \
    -X POST \
    -d '{"inputs":"What is Deep Learning?","parameters":{"max_new_tokens":20}}' \
    -H 'Content-Type: application/json'
```

You can also run queries using the Request.py file.

```
#Request.py#

import requests

url = “<endpoint>”

headers = {
    "Content-Type": "application/json",
}

data = {
    "model_id": "llama2-13b-chat-hf",
    "inputs": "what is deep learning?"
}

for i in range(100):  # Send 100 requests
    try:
        response = requests.post(url, headers=headers, json=data, verify=False)
        response.raise_for_status()  # Raise an exception for non-200 status codes

        print(response.json())  # Print the response data (assuming JSON)
    except requests.exceptions.RequestException as err:
        print("Request failed:", err)
```

This page intentionally left blank.

Chapter 8 Benchmarking Llama2 Inference Serving on AMD EPYC Systems

The latency, throughput, and scaling testing described in this user guide uses EchoSwift*.

8.1 Explaining Benchmarking

The objective of the LLM-Inference-Bench tools measures the latency of each request in milliseconds per token, Time Taken for the First Token (TTFT), and Throughput measured in the number of tokens per second of request sent to a served LLM. These metrics are captured using varying input tokens (query length), output tokens (response length), and simulated parallel users.

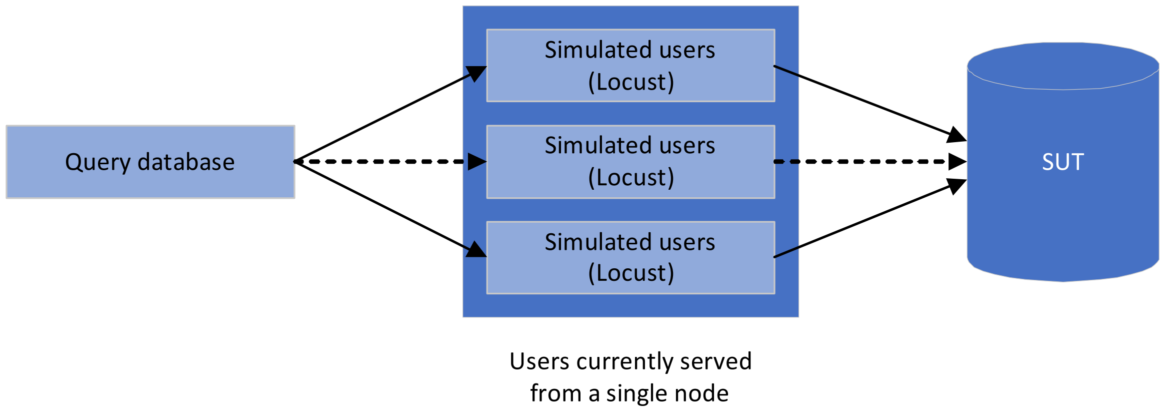


Figure 8-1: EchoSwift architecture

You can use this LLM inference benchmark to:

- **Measure performance:** Evaluate model latency, throughput, and resource (CPU-Core/Memory) utilization under varying workloads and configurations.
- **Identify bottlenecks:** Analyze performance metrics to pinpoint potential bottlenecks in different deployment scenarios impacting model efficiency and scalability.
- **Optimize deployment:** Utilize benchmarking data to fine-tune model deployment for improved resource utilization and cost-effectiveness.
- **Validate scalability:** Assess the model's ability to handle increasing workloads and maintain performance with a growing number of concurrent requests.

Metrics captured:

- Number of input tokens
- Number of output tokens
- Throughput (tokens/second)
- End-to-end latency (ms)
- Token latency (ms/tokens)
- TTFT for streaming
- CPU and memory utilization while running the load test.

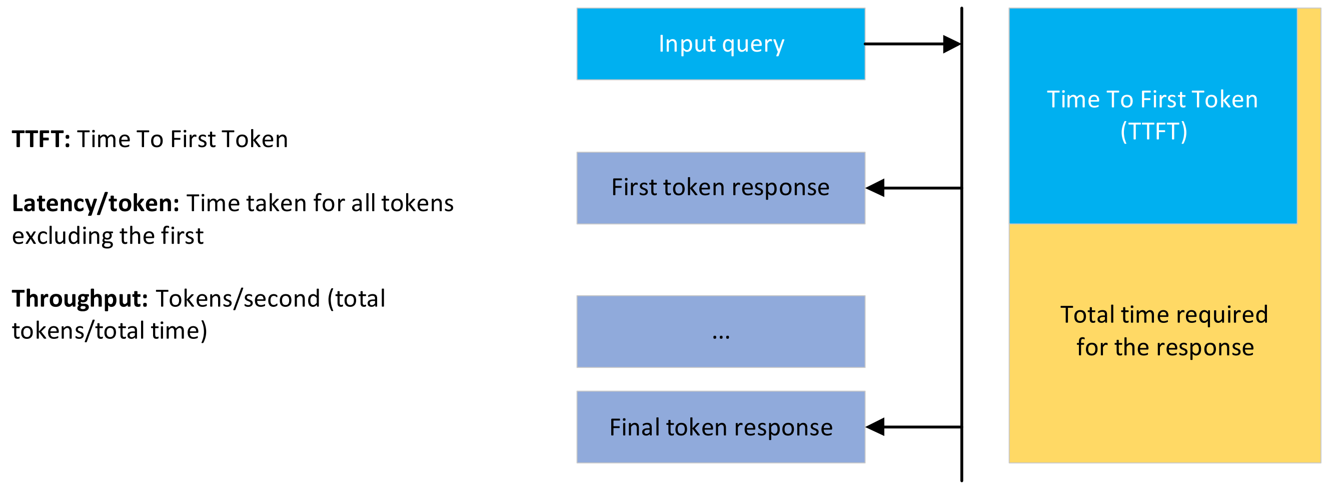

TTFT: Time To First Token
Latency/token: Time taken for all tokens excluding the first
Throughput: Tokens/second (total tokens/total time)
Input query
First token response
...
Final token response
Time To First Token (TTFT)
Total time required for the response

Figure 8-2: Latency, throughput, and TTFT calculations

Performance metrics calculations:

- **Throughput:** Measures the average number of tokens that can be generated per second:
  Throughput (tokens/Second) = (Output Token Length) / Total Time (s)
- **Token latency:** Measures the average time it takes to generate one token:
  Latency (ms/token) = Total Time (s) * 1000 / Output Token Length
- **TTFT:** Indicates the responsiveness of the model. Time from start of request to generation of first token.

8.2 Locust

Locust is an open-source tool that simulates multiuser requests on systems under test. It is built in python and excels at user behavior simulation. It is both flexible and easy of use, and its distributed testing capabilities allow simulating thousands of concurrent users. These features enable comprehensive system performance evaluation under realistic scenarios. The following prerequisites must be met in order to run the benchmark:

Item	Configuration/Version
# of cores	16
RAM	20 GB
datasets	2.13.1
locust	2.18.4
streamlit	1.29.0
docker	5.0.3
kserve	0.10.0
torch	1.13.0
transformers	4.31.0
huggingface_hub	0.20.0

Table 8-1: Locust hardware and software requirements

8.2.1 Single Docker Container Deployment

The benchmark script takes the required variables and starts sending requests to the previously-deployed model endpoint. Locust creates virtual users and simulates production-level load testing. The benchmark will run against the above generation endpoint, which can be any inference server endpoint, such as a TGI endpoint hosting a model (`http://localhost:8080/generate_stream`).

Define the configurations inside the `locust.sh` shell script before starting the load test. Here are some sample configurations:

- # of parallel users [1, 3, 10].
- Varying input tokens [32, 64, 128]
- Varying output tokens [32, 64, 128]

```
$   ./llm_inference_benchmark.sh<output_dir <generation_endpoint>
```

This page intentionally left blank.

Chapter 9 Linear Llama2 7B “CPU-Only” Scaling

Deploying LLMs for real-time inference tasks often requires efficient resource management to handle varying workloads. This chapter discusses linear scaling of CPU-only LLM serving on Llama2 7B utilizing an infrastructure built on Red Hat OpenShift. The primary objective is to showcase how the CPU resources allocated to Llama2 7B serving pods scale linearly with increasing inference requests.

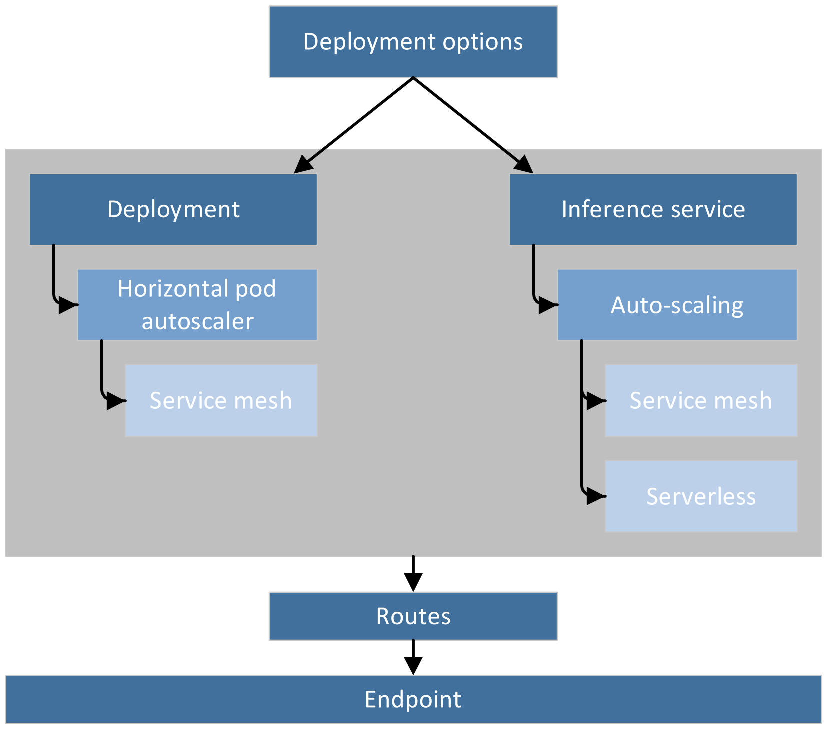


Figure 9-1: Deployment options

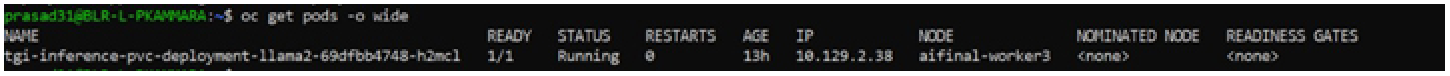
Figure 9-1 shows an OpenShift deployment using Kind for both the deployment and Inference Service. The Deployment path offers a Horizontal Pod AutoScaler to scale container instances and a Service Mesh for managing microservices. The Inference Service path provides Auto-Scaling to handle varying workloads, a Service Mesh architecture for managing services, and Serverless functions for event-driven computing as an Inference Service. OpenShift Autoscaling allows the number of serving replicas to adjust dynamically based on CPU utilization to maintain optimal performance and resource utilization. The Routes component routes and load balances incoming requests. These requests are then served through an Endpoint.

9.1 Llama-7B deployment with HPA

Attain HPA by creating a YAML with Kind deployment, 96 CPU cores, and 48 GB of memory using Horizontal Pod Scaler and then applying the following file:

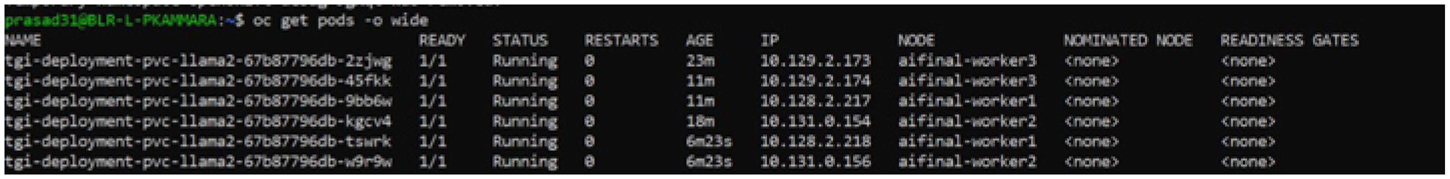
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tgi-deployment-pvc-llama2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tgi-inference
  template:
    metadata:
      labels:
        app: tgi-inference
    spec:
      containers:
        - name: tgi-deployment-pvc-container
          image: akhil2507/tgi_images:v1
          ports:
            - containerPort: 8080
          args:
            - "--model-id"
            - "meta-llama/Llama-2-7b-chat-hf"  # Replace with your actual model ID
            - "--dtype"
            - "bfloat16"
          env:
            - name: HUGGING_FACE_HUB_TOKEN
              value: <HUGGING_FACE_TOKEN> # Replace with your actual Hugging Face token
          resources:
            requests:
              memory: "48Gi"  # Adjust as needed
              cpu: 96     # Adjust as needed
            limits:
              memory: "48Gi"   # Adjust as needed
              cpu: 96    # Adjust as needed
          volumeMounts:                  # Add this section
            - name: data-volume
              mountPath: /data
      volumes:                           # Add this section
        - name: data-volume
          persistentVolumeClaim:
            claimName: llama2-pvc-deployment
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: llama2-hpa
  namespace: tgi-deployment-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: tgi-deployment-pvc-llama2
  minReplicas: 1
  maxReplicas: 4
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          averageUtilization: 50
          type: Utilization
---
apiVersion: v1
kind: Service
metadata:
  name: tgi-deployment-svc
  namespace: tgi-deployment-hpa
spec:
  selector:
    app: tgi-inference
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
---
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: tgi-deployment-route
  namespace: tgi-deployment-hpa
  labels: {}
spec:
  to:
    kind: Service
    name: tgi-deployment-svc
  tls: null
  port:
    targetPort: 8080

$ oc apply -f <filename.yaml>
```



```
prasad31@BLR-L-PKAMMARA:~$ oc get pods -o wide
NAME                                            READY   STATUS    RESTARTS   AGE   IP            NODE             NOMINATED NODE   READINESS GATES
tgi-inference-pvc-deployment-llama2-69dfbb4748-h2mcl   1/1     Running   0          13h   10.129.2.38   aifinal-worker3   <none>           <none>
```

Figure 9-2: Before automatic scaling



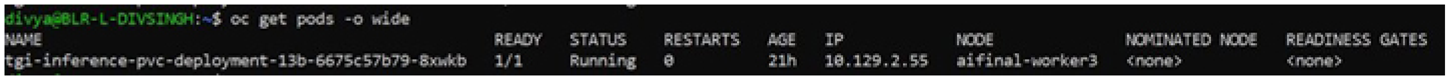
```
prasad31@BLR-L-PKAMMARA:~$ oc get pods -o wide
NAME                                        READY   STATUS    RESTARTS   AGE     IP             NODE              NOMINATED NODE   READINESS GATES
tgi-deployment-pvc-llama2-67b87796db-2zjwg   1/1     Running   0          23m     10.129.2.173   aifinal-worker3   <none>           <none>
tgi-deployment-pvc-llama2-67b87796db-45fkk   1/1     Running   0          11m     10.129.2.174   aifinal-worker3   <none>           <none>
tgi-deployment-pvc-llama2-67b87796db-9bb6w   1/1     Running   0          11m     10.128.2.217   aifinal-worker1   <none>           <none>
tgi-deployment-pvc-llama2-67b87796db-kgcv4   1/1     Running   0          18m     10.131.0.154   aifinal-worker2   <none>           <none>
tgi-deployment-pvc-llama2-67b87796db-tswrk   1/1     Running   0          6m23s   10.128.2.218   aifinal-worker1   <none>           <none>
tgi-deployment-pvc-llama2-67b87796db-w9r9w   1/1     Running   0          6m23s   10.131.0.156   aifinal-worker2   <none>           <none>
```

Figure 9-3: After automatic scaling

9.2 Llama2-13B Deployment with HPA

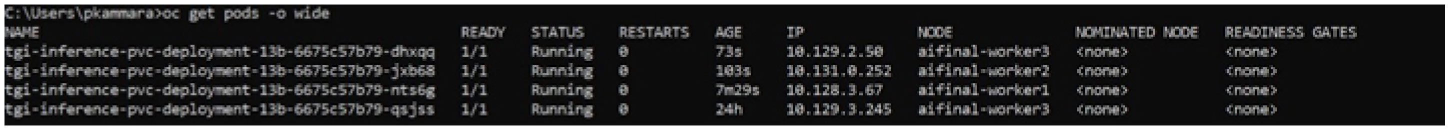
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tgi-inference-pvc-deployment-13b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tgi-inference
  template:
    metadata:
      labels:
        app: tgi-inference
    spec:
      containers:
        - name: tgi-llama2-13b-deployment-pvc-container
          image: akhil2507/tgi_images:v1
          ports:
            - containerPort: 8080
          args:
            - "--model-id"
            - "meta-llama/Llama-2-13b-chat-hf"  # Replace with your actual model ID
            - "--dtype"
            - "bfloat16"
          env:
            - name: HUGGING_FACE_HUB_TOKEN
              value: <HUGGING FACE> # Replace with your actual Hugging Face token
          resources:
            requests:
              memory: "96Gi"  # Adjust as needed
              cpu: 96     # Adjust as needed
            limits:
              memory: "96Gi"   # Adjust as needed
              cpu: 96    # Adjust as needed
          volumeMounts:                    # Add this section
            - name: data-volume
              mountPath: /data
      volumes:                            # Add this section
        - name: data-volume
          persistentVolumeClaim:
            claimName: llama2-13b-claim
---
apiVersion: v1
kind: Service
metadata:
  name: tgi-inference-pvc-deployment-13b
  namespace: llama2-13b
spec:
  selector:
    app: tgi-inference
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
---
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: tgi-inference-13b-deployment
  namespace: llama2-13b
  labels: {}
spec:
  to:
    kind: Service
    name: tgi-deployment-svc
  tls: null
  port:
    targetPort: 8080

oc apply -f <filename.yaml>
```



```
divya@BLR-L-DIVSINGH:~$ oc get pods -o wide
NAME                                            READY   STATUS    RESTARTS   AGE   IP            NODE              NOMINATED NODE   READINESS GATES
tgi-inference-pvc-deployment-13b-6675c57b79-8xwkb   1/1     Running   0          21h   10.129.2.55   aifinal-worker3   <none>           <none>
```

Figure 9-4: Before automatic scaling



```
C:\Users\pkammara>oc get pods -o wide
NAME                                              READY   STATUS    RESTARTS   AGE     IP             NODE              NOMINATED NODE   READINESS GATES
tgi-inference-pvc-deployment-13b-6675c57b79-dhxqq   1/1     Running   0          73s     10.129.2.50    aifinal-worker3   <none>           <none>
tgi-inference-pvc-deployment-13b-6675c57b79-jxb68   1/1     Running   0          103s    10.131.0.252   aifinal-worker2   <none>           <none>
tgi-inference-pvc-deployment-13b-6675c57b79-nts6g   1/1     Running   0          7m29s   10.128.3.67    aifinal-worker1   <none>           <none>
tgi-inference-pvc-deployment-13b-6675c57b79-qsjss   1/1     Running   0          24h     10.129.3.245   aifinal-worker3   <none>           <none>
```

Figure 9-5: After automatic scaling

9.3 Llama2-70B Deployment with HPA

Add the annotations to the route in order to distribute the traffic across all replicas in a round-robin-fashion:

```
$ oc annotate route <route-name> haproxy.router.openshift.io/balance=roundrobin
$ oc annotate route <route-name> haproxy.router.openshift.io/disable_cookies=true
```

This page intentionally left blank.

Mq wwsw

Chapter 10

Sizing “CPU-Only” LLM Inference Serving with an AMD EPYC 9004 System

Proposals for deploying LLMs for inferencing could have the following requirements. You may use these proposed requirements as a basis for Requests for Proposals (RFPs) for sales or internal deployment requirements. The deployed solution should support:

- 1000 parallel requests with 5+ output tokens per second at normal reading speed.
- Output token size of 256 for large answers.
- Latency under 250 ms.
- Llama2 7B BF16 like model (13.5 GB) or lower

The following considerations can help you size these requirements:

- **Worker node:** 2 x 96-core AMD EPYC 9654 processor, 384 GB memory, 10 Gbps network, 4 x 1 TB SSD storage
- **Model replica size:** 96 cores, 48 GB memory, with Simultaneous Multi Threating (SMT) enabled (supports 7-8 replicas per node). 8 replicas configured as shown here can process 30 parallel requests with latency less than 250 ms for 256 output tokens with near-linear scaling.
- **Number of replicas/nodes needed:** 1000 parallel requests can be catered by 1000/30 or 34 Replicas, which translated to 11 nodes, assuming 20 cores per node for non-replica usage where each node can deploy 3 x 96-core replicas.

This page intentionally left blank.

Chapter 11

Appendix (Tech-Preview Features)

11.1 Tech-Preview CPU Optimizations with CPU Manager Policy

CPU Manager Policies has many enhancements, such as `tech-preview` (do not test this production), which provides additional functionality around how cores are used by pods and how replica sets are selected. These are available as alpha and beta options with Kubernetes version 1.22 or higher. The following section describes how to set these policies in a test environment for evaluation. Please see Control CPU Management Policies on the Node* for details about these options.

- `full-pcpus-only` is a beta that is visible by default in Kubernetes 1.22 or higher.
- `distribute-cpus-across-numa` is an alpha that is hidden by default in Kubernetes 1.23 or higher.
- `align-by-socket` is an alpha that is hidden by default in Kubernetes 1.25 or higher.

Note: Tech-Preview features are for testing and evaluating new features. Do not deploy them in production

11.2 Setting Feature-Gate Options (Enabling Alpha/Beta Options)

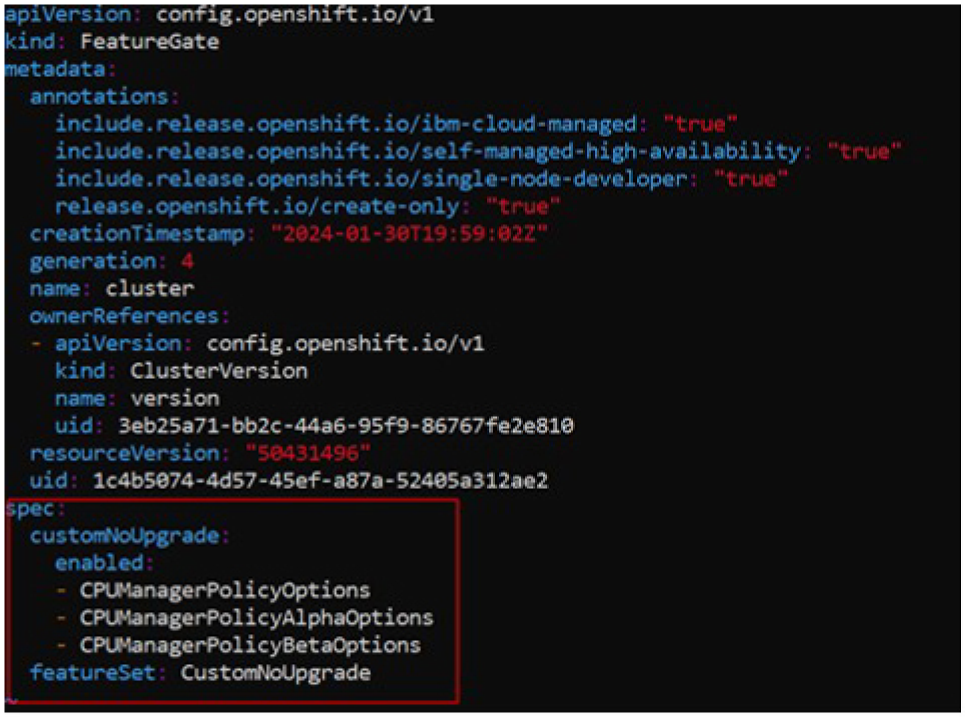
Feature gates are a set of keys (opaque string values) that you can use to control which features are enabled in the cluster. Each feature gate is a set of `key=value` pairs that describe Kubernetes features. You can enable or disable these features using the `--feature-gates` command line flag on each Kubernetes component. The following prerequisites must be met:

- Ensure that the CPU Manager is enabled (`cpumanager-enabled`).
- Configure the `static` CPU Manager policy.

To edit feature gates:

1. Execute the following command to add the required feature gates to enable CPU Manager Policy options:

```
$ oc edit featuregate cluster
```



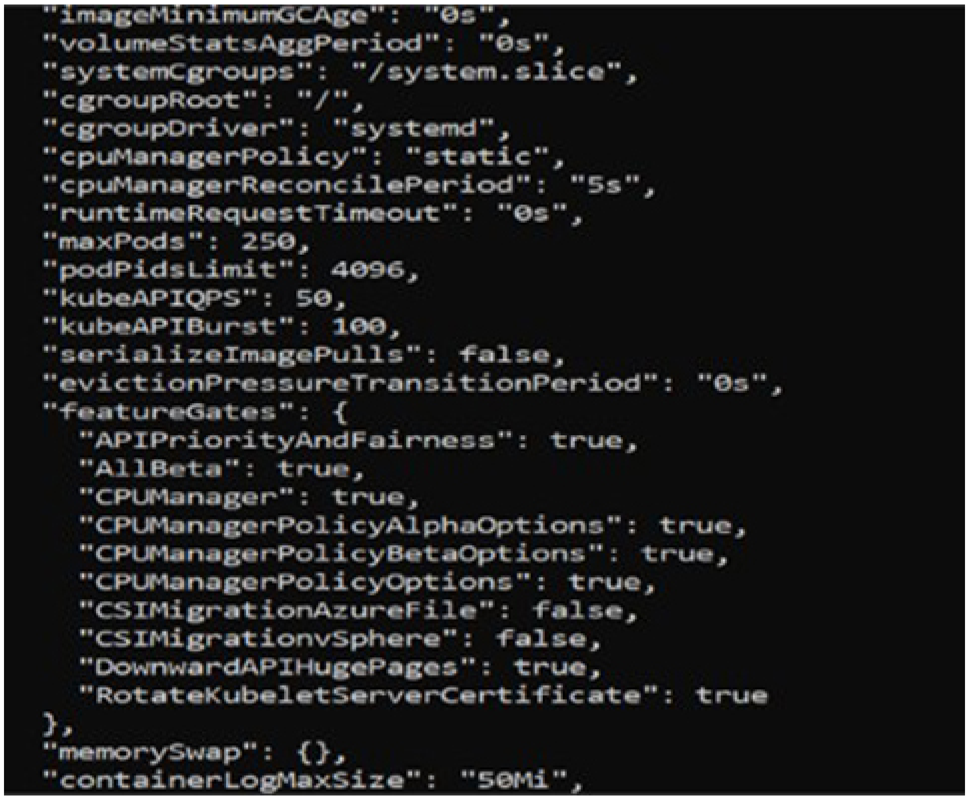
```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  annotations:
    include.release.openshift.io/ibm-cloud-managed: "true"
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
    release.openshift.io/create-only: "true"
  creationTimestamp: "2024-01-30T19:59:02Z"
  generation: 4
  name: cluster
  ownerReferences:
  - apiVersion: config.openshift.io/v1
    kind: ClusterVersion
    name: version
    uid: 3eb25a71-bb2c-44a6-95f9-86767fe2e810
  resourceVersion: "50431496"
  uid: 1c4b5074-4d57-45ef-a87a-52405a312ae2
spec:
  customNoUpgrade:
    enabled:
    - CPUManagerPolicyOptions
    - CPUManagerPolicyAlphaOptions
    - CPUManagerPolicyBetaOptions
  featureSet: CustomNoUpgrade
```

Figure 11-1: Editing feature gates

2. Verify that the feature gates are enabled in each node.

```
$ oc debug node/<node-name>
$ chroot /host
$ cat /etc/kubernetes/kubelet.conf
```

The output appears as follows:



```
"imageMinimumGCAge": "0s",
"volumeStatsAggPeriod": "0s",
"systemCgroups": "/system.slice",
"cgroupRoot": "/",
"cgroupDriver": "systemd",
"cpuManagerPolicy": "static",
"cpuManagerReconcilePeriod": "5s",
"runtimeRequestTimeout": "0s",
"maxPods": 250,
"podPidsLimit": 4096,
"kubeAPIQPS": 50,
"kubeAPIBurst": 100,
"serializeImagePulls": false,
"evictionPressureTransitionPeriod": "0s",
"featureGates": {
  "APIPriorityAndFairness": true,
  "AllBeta": true,
  "CPUManager": true,
  "CPUManagerPolicyAlphaOptions": true,
  "CPUManagerPolicyBetaOptions": true,
  "CPUManagerPolicyOptions": true,
  "CSIMigrationAzureFile": false,
  "CSIMigrationvSphere": false,
  "DownwardAPIHugePages": true,
  "RotateKubeletServerCertificate": true
},
"memorySwap": {},
"containerLogMaxSize": "50Mi",
```

Figure 11-2: Sample output

3. Apply the custom kubelet config with CPU Manager policy options:

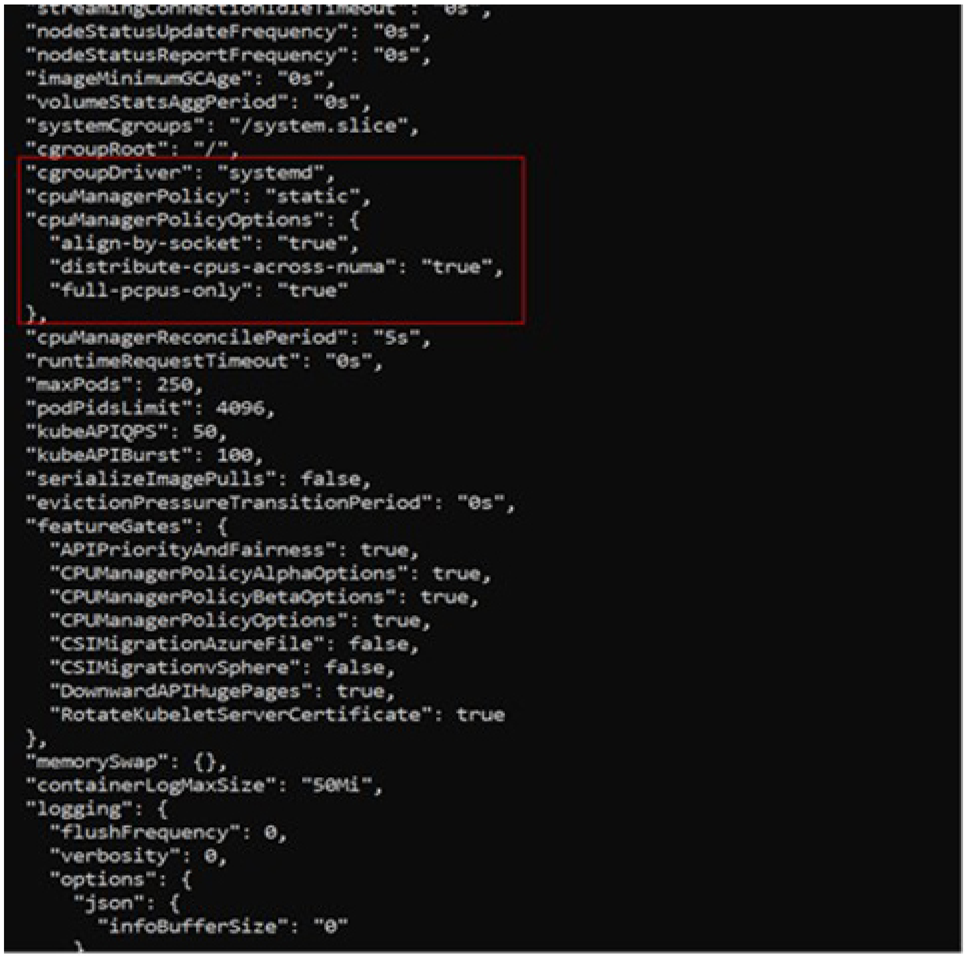
```
$ oc apply -f custom-kubeletconfig.yaml

apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
     cpuManagerPolicy: static
     cpuManagerReconcilePeriod: 5s
     cpuManagerPolicyOptions:
       align-by-socket: "true"
       distribute-cpus-across-numa: "true"
       full-pcpus-only: "true"
```

4. Verify that the `cpuManagerPolicies` are applied:

```
$ oc debug node/<node-name>
$ chroot /host
$ cat /etc/kubernetes/kubelet.conf
```

The output appears as follows:



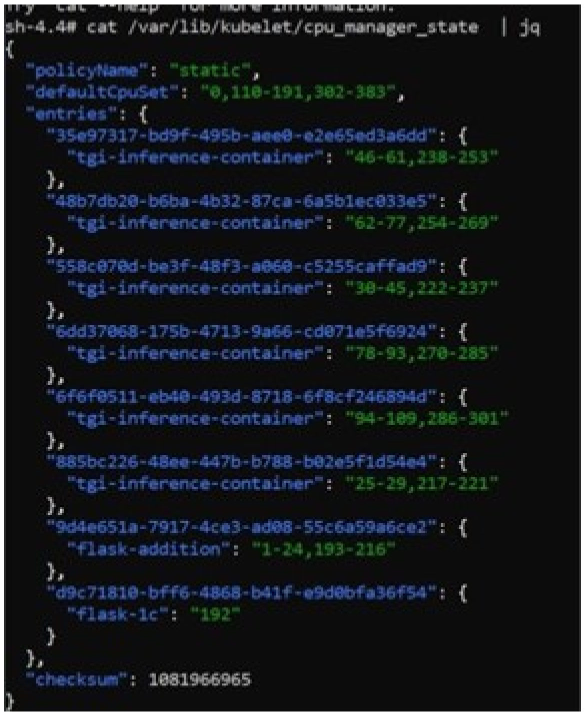
```
"streamingConnectionIdleTimeout": "0s",
"nodeStatusUpdateFrequency": "0s",
"nodeStatusReportFrequency": "0s",
"imageMinimumGCAge": "0s",
"volumeStatsAggPeriod": "0s",
"systemCgroups": "/system.slice",
"cgroupRoot": "/",
"cgroupDriver": "systemd",
"cpuManagerPolicy": "static",
"cpuManagerPolicyOptions": {
  "align-by-socket": "true",
  "distribute-cpus-across-numa": "true",
  "full-pcpus-only": "true"
},
"cpuManagerReconcilePeriod": "5s",
"runtimeRequestTimeout": "0s",
"maxPods": 250,
"podPidsLimit": 4096,
"kubeAPIQPS": 50,
"kubeAPIBurst": 100,
"serializeImagePulls": false,
"evictionPressureTransitionPeriod": "0s",
"featureGates": {
  "APIPriorityAndFairness": true,
  "CPUManagerPolicyAlphaOptions": true,
  "CPUManagerPolicyBetaOptions": true,
  "CPUManagerPolicyOptions": true,
  "CSIMigrationAzureFile": false,
  "CSIMigrationvSphere": false,
  "DownwardAPIHugePages": true,
  "RotateKubeletServerCertificate": true
},
"memorySwap": {},
"containerLogMaxSize": "50Mi",
"logging": {
  "flushFrequency": 0,
  "verbosity": 0,
  "options": {
    "json": {
      "infoBufferSize": "0"
    }
```

Figure 11-3: CPU Manager policy verification

5. Verify the CPU Manager policy state:

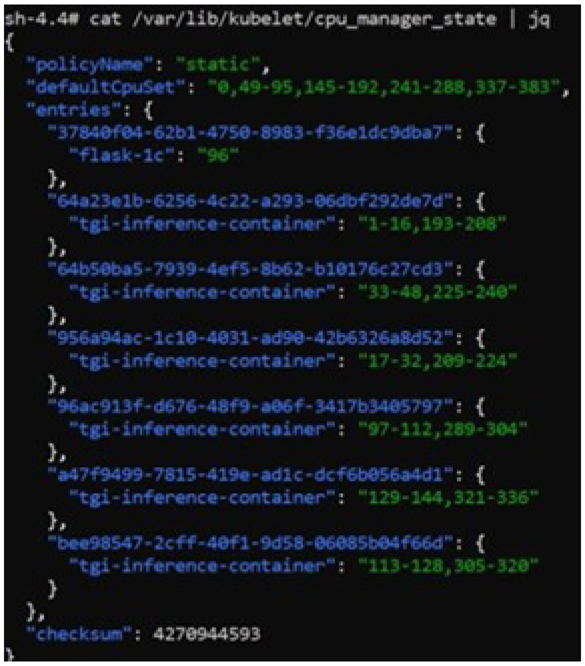
```
$ oc debug node/<node-name>
$ chroot /host
$ cat /var/lib/kubelet/cpu_manager_state | jq
```

The output appears as follows:



```
Try 'cat --help' for more information.
sh-4.4# cat /var/lib/kubelet/cpu_manager_state  | jq
{
  "policyName": "static",
  "defaultCpuSet": "0,110-191,302-383",
  "entries": {
    "35e97317-bd9f-495b-aee0-e2e65ed3a6dd": {
      "tgi-inference-container": "46-61,238-253"
    },
    "48b7db20-b6ba-4b32-87ca-6a5b1ec033e5": {
      "tgi-inference-container": "62-77,254-269"
    },
    "558c070d-be3f-48f3-a060-c5255caffad9": {
      "tgi-inference-container": "30-45,222-237"
    },
    "6dd37068-175b-4713-9a66-cd071e5f6924": {
      "tgi-inference-container": "78-93,270-285"
    },
    "6f6f0511-eb40-493d-8718-6f8cf246894d": {
      "tgi-inference-container": "94-109,286-301"
    },
    "885bc226-48ee-447b-b788-b02e5f1d54e4": {
      "tgi-inference-container": "25-29,217-221"
    },
    "9d4e651a-7917-4ce3-ad08-55c6a59a6ce2": {
      "flask-addition": "1-24,193-216"
    },
    "d9c71810-bff6-4868-b41f-e9d0bfa36f54": {
      "flask-1c": "192"
    }
  },
  "checksum": 1081966965
}
```

Figure 11-4: Cores allocated across sockets hefore enabling CPU Manager policy options



```
sh-4.4# cat /var/lib/kubelet/cpu_manager_state | jq
{
  "policyName": "static",
  "defaultCpuSet": "0,49-95,145-192,241-288,337-383",
  "entries": {
    "37840f04-62b1-4750-8983-f36e1dc9dba7": {
      "flask-1c": "96"
    },
    "64a23e1b-6256-4c22-a293-06dbf292de7d": {
      "tgi-inference-container": "1-16,193-208"
    },
    "64b50ba5-7939-4ef5-8b62-b10176c27cd3": {
      "tgi-inference-container": "33-48,225-240"
    },
    "956a94ac-1c10-4031-ad90-42b6326a8d52": {
      "tgi-inference-container": "17-32,209-224"
    },
    "96ac913f-d676-48f9-a06f-3417b3405797": {
      "tgi-inference-container": "97-112,289-304"
    },
    "a47f9499-7815-419e-ad1c-dcf6b056a4d1": {
      "tgi-inference-container": "129-144,321-336"
    },
    "bee98547-2cff-40f1-9d58-06085b04f66d": {
      "tgi-inference-container": "113-128,305-320"
    }
  },
  "checksum": 4270944593
}
```

Figure 11-5: Cores allocated from the same NUMA node and and same socket after enabling CPU Manager policy (ALPHA) options

This page intentionally left blank.