Cloudera AI Inference (Preview)

Date published: 2024-05-16
Date modified: 2024-05-16

Legal Notice

This document has been released as part of a technical preview for features described herein. Technical preview components are provided as a convenience to our customers for their evaluation and trial usage. These components are provided ‘as is’ without warranty or support. Further, Cloudera assumes no liability for the usage of technical preview components, which should be used by customers at their own risk.

Contents

Legal Notice **2**
Contents **3**
About this feature **6**
Introduction 6
Key Features 6
Limitations and Restrictions 7
Product Version 7
API Stability 7
Cloud Platforms 7
Supported Instance Types 7
No Support for Heterogeneous GPUs 8
Supported Model Artifact Formats 8
Large Language Models 8
Supported TensorRT LLM Models 8
Supported Transformer Models from Hugging Face 10
CLI Only Interface 10
No Private Cluster and Non-Transparent Proxy Support 10
Public Load Balancer 10
Load Balancer Ingress Whitelist CIDR 11
Monitoring 11
Logging 11
Prerequisites **11**
CDP Entitlement 11
Authentication 11
Authorization 12
CML Registry 12
Register an ONNX model to CML Registry 12
Register a TRT-LLM model to CML Registry 13
Register a Hugging Face Transformer model to CML Registry 13
CDP Control Plane Operations **15**
Creating a Cloudera AI Inference App 15
AWS 16
Azure 18

Listing Cloudera AI Inference Apps 19
Describing Cloudera AI Inference App 20
Deleting Cloudera AI Inference App 20
Cloudera AI Inference Workload Operations 21
Interacting With Cloudera AI Inference 21
REST Client 21
Preparing to Interact with the Cloudera AI Inference API 21
Creating a Model Endpoint 22
Listing Model Endpoints 23
Describing a Model Endpoint 23
Making an inference call to a Model Endpoint 24
Open Inference Protocol 24
OpenAI Inference Protocol 26
Deleting a Model Endpoint 27
Autoscaling Model Endpoints 27
Tuning Autoscaling Sensitivity 28
Running Models on GPU 29
Deploying Models with Canary Deployment 29
GUI Client 31
Listing Model Endpoints 32
Creating a Model Endpoint 33
Viewing Model Endpoint Details 34
Editing a Model Endpoint Configuration 35
How Do I 37
Deploying a TRT-LLM Optimized Large Language Model 37
Deploying a Hugging Face Transformer Large Language Model 48
Deploying a Predictive Deep Learning Model 60
Obtaining the kubeconfig for my Cloudera AI Inference Service Cluster 65
AWS 65
Azure 66
Known issues and limitations 67
Manually Updating Model Registry Configuration 68
1. Get KubeConfig for AI Inference Service Cluster 68
2. Get New ConfigMap Data 68
4. Apply ConfigMap Update 71

About this feature

Introduction

The Cloudera AI Inference service provides a production-grade serving environment for hosting predictive, generative AI, and large language models (LLMs). It is designed to handle the challenges of production deployments, such as high availability, performance, fault tolerance, and scalability. The Cloudera AI Inference service allows data scientists and machine learning engineers to deploy their models quickly, without worrying about the infrastructure and maintenance.

Cloudera AI Inference is built with tight integration of NVIDIA NeMo Inference Microservices (NIMs), including NVIDIA Triton Inference Server, NeMo Inference Server for TensorRT LLMs, and NVIDIA NIM for LLMs, providing industry-leading inference performance on NVIDIA GPUs. Model endpoint orchestration and management is built with the KServe model inference platform, a CNCF open source project. The platform provides standards-compliant model inference protocols, such as Open Inference Protocol for predictive models and OpenAI API for generative AI models.

Cloudera AI Inference service is seamlessly integrated with the CML Registry standalone API enabling users to store, manage, and track their machine learning models throughout their lifecycle. This integration provides a central location to store model and application artifacts, metadata, and versions, making it easier to share and reuse ML artifacts across different teams and projects. It also simplifies the process of deploying models and applications to production, as users can select artifacts from the registry and deploy them with a single command.

Key Features

Key features of Cloudera AI Inference include:

- **Easy to use interface**: This streamlines the complexities of deployment and infrastructure, meaningfully reducing time to value for AI use cases.
- **Real-time predictions**: Cloudera AI Inference allows users to serve machine learning models in real-time, providing low latency predictions for client requests.
- **Monitoring and logging**: Cloudera AI Inference includes functionality for monitoring and logging, making it easier to troubleshoot issues and optimize workload performance.

- **Advanced deployment patterns**: Cloudera AI Inference includes functionality for advanced deployment patterns, such as canary and blue-green deployments, and supports A/B testing, enabling users to deploy new versions of models gradually and compare their performance before deploying them to production.
- **Hardware acceleration**: Cloudera AI Inference integrates with NVIDIA to accelerate inference performance on GPU cards.
- **Model access**: Cloudera AI Inference offers access to NVIDIA NeMo models and optimized open-source models, tailored for NVIDIA hardware to increase inference throughput and reduce latency.
- **REST API**: Cloudera AI Inference provides APIs for deploying, managing, and monitoring online models. These APIs enable integration with CI/CD pipelines and other tools used in the MLOps and LLMOps workflows.
- **Command Line Interface**: A CLI is provided to allow ease of use and scriptability. The CLI is built from the REST API specification and provides the same capabilities.

Limitations and Restrictions

Product Version

The only valid version when creating a Cloudera AI Inference App instance is “1.1.0-b53”.

API Stability

Both the ML Control Plane and Cloudera AI Inference workload APIs and CLIs are under active development and are subject to change in backwards-incompatible ways.

Cloud Platforms

Cloudera AI Inference is available on AWS and Azure only. Private Cloud support is planned for the future.

Supported Instance Types

Cloudera AI Inference supports the same cloud instance types as those for CML workspaces. The type or size of the model you want to deploy determines the cloud compute instance type. Some highly optimized versions of Large Language Models, for instance, will only work on specific GPU architectures.

No Support for Heterogeneous GPUs

The tech preview release only supports two compute worker node groups: one each for CPU and GPU. Heterogeneous GPU support is planned for the future.

Supported Model Artifact Formats

The current version of Cloudera AI Inference supports the following models:

- ONNX models registered to CML Registry from the CML Workspace. Here is an example showing how to convert a sklearn model to ONNX and then registering it to the CML Registry. Refer to PyTorch or TensorFlow documentation regarding how models using these frameworks can be converted to the ONNX representation.
- TensorRT LLM models for text generation from NVIDIA GPU Cloud (NGC) Catalog generated for version 24.01 of the Nemo Inference Microservice.
- Transformer models from Hugging Face for text generation with model architectures in this table.

Large Language Models

The technical preview version of Cloudera AI Inference supports the deployment of LLM endpoints. The LLM artifacts must be imported to a CML Registry running in the same CDP environment before it can be deployed to the Cloudera AI Inference service.

Supported TensorRT LLM Models

The following large language models have been optimized for TRT LLM by NVIDIA, and are known to work with Cloudera AI Inference service.

Model Name	Short Description	Repository Path Prefix ohlfw0olaadg/ea-participants/	GPU/GPU Memory/Num GPUs	Example GPU Instance Types
Llama-2-7b	Llama 2 7B is one of a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.	llama-2-7b	A100/80G/1	p4d.24xlarge Standard_NC24ads_A100_v4
Llama-2-7b	Llama 2 7B is one of a	llama-2-7b-chat	A100/80G/	p4d.24xlarge

-Chat	collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.		1	Standard_NC24ads_A100_v4
Llama2-13b	Llama 2 13B is one of a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.	llama-2-13b	A100/80G/1	p4d.24xlarge Standard_NC24ads_A100_v4
Llama2-13b-Chat	Llama 2 13B is one of a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.	llama-2-13b-chat	A100/80G/1	p4d.24xlarge Standard_NC24ads_A100_v4
Llama2-70b	Llama 2 70B is one of a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.	llama-2-70b	A100/80G/4	p4d.24xlarge Standard_NC96ads_A100_v4
Llama2-70b-Chat	Llama 2 70B is one of a collection of pre-trained and fine-tuned generative text models ranging in scale from 7 billion to 70 billion parameters developed by Meta.	llama-2-70b-chat	A100/80G/4	p4d.24xlarge Standard_NC96ads_A100_v4
StarCoder	The StarCoder2-15B model is a 15B parameter model trained on 600+ programming languages from The Stack v2, with opt-out requests excluded.	starcoder	A100/80G/1	p4d.24xlarge Standard_NC24ads_A100_v4
StarCoder	The StarCoder2-15B	starcoderplus	A100/80G/	p4d.24xlarge

Plus	model is a 15B parameter model trained on 600+ programming languages from The Stack v2, with opt-out requests excluded.		1	Standard_NC24ads_A100_v4
NV-GPT-43B-instruct	NV-GPT-43B-instruct model published by NVIDIA	FP16/4K	A100/80G/2	p4d.24xlarge Standard_NC48ads_A100_v4
NV-GPT-43B-chat	NV-GPT-43B-chat model published by NVIDIA	FP16/4K	A100/80G/2	p4d.24xlarge Standard_NC48ads_A100_v4

Supported Transformer Models from Hugging Face

Text generation transformer models from Hugging Face are supported if the model architecture is supported by the vLLM inference engine. Supported architectures and popular models that use each architecture are shown in this Supported Models table. In general, all LLMs run best on GPUs. We recommend at least NVIDIA A10 or newer GPUs for deploying text generation LLMs.

CLI Only Interface

The technical preview release provides CDP CLI user interface for control plane operations. The ***beta*** version of CDP CLI, version 0.9.113 or higher, must be used to manage a Cloudera AI Inference App instance or cluster for the current release. Refer to the CDP CLI Beta documentation for installation instructions.

The workload provides a set of REST APIs and a GUI to manage model endpoints.

No Private Cluster and Non-Transparent Proxy Support

Cloudera AI Inference has not been tested in a private cluster configuration, nor has it been tested with a non-transparent proxy (NTP) setup.

Public Load Balancer

By default, Cloudera AI Inference will use a private load balancer for cluster ingress. If you want to use a public load balancer instead, set the `usePublicLoadBalancer` parameter to `true` in the creation payload. For more information, see Creating a Cloudera AI Inference App.

Load Balancer Ingress Whitelist CIDR

This version of Cloudera AI Inference service does not support specification of IP addresses or address ranges that are allowed to connect to the ingress load balancer.

Monitoring

Cloudera AI Inference service exposes Prometheus metrics for deployed model endpoints. The GUI displays plots of a few chosen metrics for each model endpoint. For additional metrics, the Prometheus server can also be accessed directly at https://${DOMAIN_NAME}/prometheus by passing the authorization bearer token. For more information, see Authentication.

Logging

All Kubernetes pod logs, including pods that are running model servers, are scraped by the platform log aggregator service (fluentd). The technical preview release does not provide a mechanism to view these logs on the GUI or API. To view the logs, you must first obtain the kubeconfig of the cluster and use the kubectl command. This also means that the historical pod logs cannot be conveniently accessed at this time.

Prerequisites

CDP Entitlement

The new data service is gated behind the `ML_ENABLE_ML_SERVING` entitlement. This entitlement must be granted to your CDP account before you attempt to create a Cloudera AI Inference App or Service instance.

Authentication

Clients that interact with the Cloudera AI Inference API and with model endpoints must obtain a JSON Web Token (JWT) from the CDP control plane, which must be passed as a Bearer token in HTTP requests sent to the serving API and endpoints. You can obtain the JWT using CDP CLI as follows:

```
$ CDP_TOKEN=$(cdp iam generate-workload-auth-token --workload-name DE
| jq -r '.token')
```

Then pass CDP_TOKEN in the HTTP request header as follows

```
$ curl -H "Authorization: Bearer ${CDP_TOKEN}" <URL>
```

The token obtained using this method expires in one hour.

Authorization

Cloudera AI Inference implements role-based access control. It allows access for CDP users with the following IAM roles:

- MLUser (Workload only)
- MLAdmin (Control Plane and Workload)

Deployed model endpoints can be accessed by users with MLUser or MLAdmin roles.

CML Registry

A CML model registry must be deployed in the same CDP environment in which you are planning to deploy Cloudera AI Inference. That is, the Cloudera AI Inference application can only deploy models logged to a CML Registry in the same environment. For this release, the model registry version *CML2024.04-2* or higher must be created before provisioning the Cloudera AI Inference service. If you have an existing CML Registry in the environment, you must first upgrade it to version CML2024.04-2 before deploying Cloudera AI Inference service. For more information on how to create a CML Registry, see Setting up Model Registry.

Register an ONNX model to CML Registry

Refer to Deploying a Predictive Deep Learning Model for an example of how to register an ONNX model to CML Registry.

CML workspace also supports registering the models from the UI. For more information, see Registering a model using the Model Registry user interface.

Register a TRT-LLM model to CML Registry

You must first obtain the model registry's domain. To list the available CML Registries, run the following:

```
$ cdp ml list-model-registries
```

From the resulting list choose the model registry that is in the same environment as your Cloudera AI Inference cluster. The registry's domain can be found in the `domain` field.

```
$ MR_DOMAIN=<domain of the registry>
```

To register a model from NGC for example, execute the following query:

```
$ curl -v -H "Content-Type: application/json" \
-H "Authorization: Bearer ${CDP_TOKEN}" \
"${MR_DOMAIN}/api/v2/models" \
-d '{
  "name": "llama2_7b_chat_from_ngc",
  "createModelVersionRequestPayload": {
    "metadata": {"model_repo_type": "NGC"},
    "downloadModelRepoRequest": {
      "source": "NGC",
      "repo_id":
"ohlfw0olaadg/ea-participants/llama-2-7b-chat:LLAMA-2-7B-CHAT-4K-FP16
-1-A100.24.01"
    }
  }
}'
```

Register a Hugging Face Transformer model to CML Registry

To register a model from Hugging Face, execute the following query:

```
$ curl -v -H "Content-Type: application/json" \
-H "Authorization: Bearer ${CDP_TOKEN}" \
"${MR_DOMAIN}/api/v2/models" \
-d '{
  "name": "llama2_7b_chat_awq_from_hf",
  "createModelVersionRequestPayload": {
    "metadata": {
```

```
      "model_repo_type": "HF"
    },
    "downloadModelRepoRequest": {
      "source": "HF",
      "repo_id": "TheBloke/Llama-2-7B-Chat-AWQ"
    }
  }
}'
```

The response from CML registry will look like this:

```
{"models":
  [
    {
      "created_at":"2024-05-02T12:58:23.293Z",
      "creator":{"user_name":"csso_user"},
      "id":"a4wj-kbhy-7y8z-ircp",
      "name":"llama2_7b_chat_awq_from_hf",
      "tags":null,
      "updated_at":"2024-05-02T12:58:23.293Z",
      "visibility":"private"
    }
  ]
}
```

To inspect different versions of the model registered in CML Registry, run the following:

```
$ curl -v -H "Content-Type: application/json" \
-H "Authorization: Bearer ${CDP_TOKEN}" \
"${MR_DOMAIN}/api/v2/models/<model_id>
```

The response looks like this:

```
{
  "created_at":"2024-04-30T19:05:44.183Z",
  "creator":{"user_name":"csso_user"},
```

```
  "id":"x33p-j71a-sox3-kmtl",
  "model_versions":[
    {
      "created_at":"2024-04-30T19:05:44.184Z",
      "metadata":{...},
      "model_id":"x33p-j71a-sox3-kmtl",
      "status":"READY",
      "tags":null,
      "updated_at":"2024-04-30T19:05:44.184Z",
      "user":{"user_name":"csso_user"},
      "Version":1
    }],
  "name":"ElasticnetWineModel",
  "tags":null,
  "updated_at":"2024-04-30T19:05:44.183Z",
  "visibility":"private"
}
```


A model can have multiple versions registered. Only model versions with the status “READY” can be deployed from CML Registry.

CDP Control Plane Operations

Creating a Cloudera AI Inference App

You must download the ***beta*** version of CDP CLI and install it to create Cloudera AI Inference App. For information on installing the CLI, see Installing Beta CDP CLI. Version 0.9.113 or higher is required for Cloudera AI Inference service.

> **Important**
>
> The shape - the compute worker node group configuration - of the Cloudera AI Inference service cluster and the compute instance types to choose from are highly dependent on the kind of model you wish to deploy. Refer to the Deploying Models section below for some guidelines and examples.

Create the Cloudera AI Inference App by invoking the following command. The recommended way to create a Cloudera AI Inference App is by first generating the CLI skeleton, customizing

the JSON file, and then passing the file to the creation command, as shown in the following example:

```
$ cdp ml create-ml-serving-app --generate-cli-skeleton >
/tmp/create-serving-app-input.json
```

Then customize the JSON file to look something like the following:

AWS

```
{
    "appName": "my-cml-serving-app",
    "environmentCrn": "Your-CDP-environment-CRN",
    "isPrivateCluster": false,
    "provisionK8sRequest": {
        "instanceGroups": [
            {
                "instanceType": "m5.4xlarge",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 256
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 5,
                    "enabled": true
                }
            },
            {
                "instanceType": "p3.8xlarge",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 5,
```

```
                    "enabled": true
                }
            }
        ],
        "environmentCrn": "Your-CDP-environment-CRN",
        "tags": [
            {
                "key": "experience",
                "value": "cml-serving"
            }
        ]
    },
    "mlservingVersion": "1.1.0-b53",
    "usePublicLoadBalancer": false
}
```

Limitation

On AWS, there is a limitation on the instance group parameter. The instanceGroups in provisionK8sRequest must follow the following pattern

- One CPU node group

```
        "instanceGroups": [
            {
                "instanceType": "m5.4xlarge",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 256
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 5,
                    "enabled": true
                }
            }
        ]
```

- One CPU node group followed by one GPU node group (same as the example above). The CPU node group needs to be specified first in the input payload.
- GPU-only node group configuration is not supported.

Azure

```
{
    "appName": "my-cml-serving-app",
    "environmentCrn": "Your-CDP-environment-CRN",
    "isPrivateCluster": false,
    "provisionK8sRequest": {
        "instanceGroups": [
            {
                "instanceType": "Standard_D4s_v3",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 256
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 5,
                    "enabled": true
                }
            },
            {
                "instanceType": "Standard_NC12s_v3",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 5,
                    "enabled": true
                }
            }
        ],
        "environmentCrn": "Your-CDP-environment-CRN",
```

```
        "tags": [
            {
                "key": "experience",
                "value": "cml-serving"
            }
        ]
    },
    "mlservingVersion": "1.1.0-b53",
    "usePublicLoadBalancer": false
}
```

Now feed the JSON file to the creation command:

```
$ cdp ml create-ml-serving-app --cli-input-json
file:///tmp/create-serving-app-input.json
```

A successful invocation of the creation command will return the CRN of the Cloudera AI Inference App instance that is being created. The command provisions a Kubernetes cluster and installs the necessary software components.

Listing Cloudera AI Inference Apps

Use the following command to list Cloudera AI Inference App instances in your CDP tenant:

```
$ cdp ml list-ml-serving-apps
```

A Cloudera AI Inference App instance that has been created successfully should show the status as “installation:finished” similar to the following:

```
{
  "appName": "peter-ml-serving-test",
  "appCrn":
"crn:cdp:ml:us-west-1:9d74eee4-1cad-45d7-b645-7ccf9edbb73d:mlserving:
9e0ea4a8-ac76-44d0-9c48-b56239525908",
  "environmentCrn":
"crn:cdp:environments:us-west-1:9d74eee4-1cad-45d7-b645-7ccf9edbb73d:
```

```
environment:ddbfdabe-e955-4000-b2c2-5dcef07759e4",
  "environmentName": "eng-ml-dev-env-azure",
  "ownerEmail": "pjiang@cloudera.com",
  "mlServingVersion": "1.1.0-b53",
  "isPrivateCluster": false,
  "creationDate": "2024-01-17T22:57:38.682000+00:00",
  "cluster": {
    "clusterName": "ml-15301043-964",
    "domainName": "ml-15301043-964.eng-ml-d.xcu2-8y8x.dev.cldr.work",
    "liftieID": "liftie-st8fv9m6",
    "isHttps": true,
    "isPublic": false,
    "ipAllowlist": "0.0.0.0/0",
    "authorizedIpRangesAllowList": false
  },
  "status": "installation:finished"
 }
```

Describing Cloudera AI Inference App

You can “describe” a Cloudera AI Inference App using the following command:

```
$ cdp ml describe-ml-serving-app --app-crn <app-crn>
```

Deleting Cloudera AI Inference App

A CML Serving App can be deleted as follows:

```
$ cdp ml delete-ml-serving-app --app-crn <app-crn>
```

Cloudera AI Inference Workload Operations

Cloudera AI Inference lets you deploy models saved in the CML Registry. The supported set of model artifact formats are described in the section Model Artifact Format.

Interacting With Cloudera AI Inference

Cloudera AI Inference provides HTTP/REST API over HTTPS and a command line interface to manage model endpoint deployments, with a graphical user interface accessible from the CML Control Plane.

REST Client

An HTTP/REST client, such as curl, can be used to interact with the CML Serving App.

Preparing to Interact with the Cloudera AI Inference API

You will need the domain name of the Cloudera AI Inference App and your CDP JSON Web Token (JWT). Get the domain name from the output of `list-ml-serving-apps` or `describe-ml-serving-app` CDP commands and store it in an environment variable. For example:

```
$ export DOMAIN=$(cdp ml describe-ml-serving-app --app-crn <app-crn> | jq
-r '.app.cluster.domainName')
```

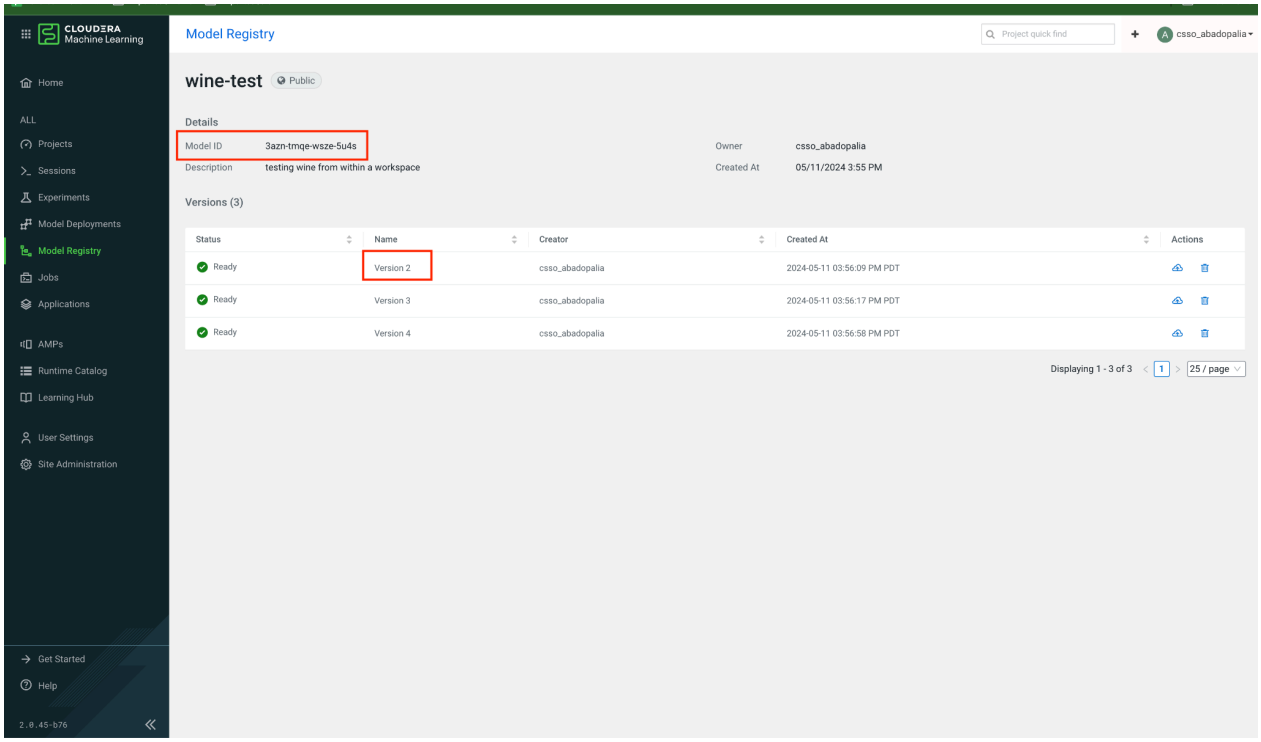
Next, store your CDP_TOKEN as an environment variable:

```
$ export CDP_TOKEN=$(cdp iam generate-workload-auth-token --workload-name
DE | jq -r '.token')
```

Note: the above command uses the `jq` command, which must be installed on your system.

Creating a Model Endpoint

To create a model endpoint, you need to retrieve a registered **model ID**, and **model version**,. This information can be found in the Model Registry view in the CML workspace UI or retrieved through the CML Registry REST API.



Create the model spec for the selected model.

```
# cat ./examples/mlflow/model-spec-cml-registry.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 2,
      "model_id": "3azn-tmqe-wsze-5u4s"
    }
  }
}
```

Create the model endpoint through CML serving `deployEndpoint` API

```
curl -v  -H "Content-Type: application/json" -H "Authorization:
Bearer ${CDP_TOKEN}" "https://${DOMAIN}/api/v1alpha1/deployEndpoint"
```

```
-d @./examples/mlflow/model-spec-cml-registry.json
```

where the DOMAIN should look like ml-67814ad5-b79.eng-ml-d.xcu2-8y8x.dev.cldr.work and CDP_TOKEN can be retrieved by following the doc here.

Note: At this time you can only specify “serving-default” as the namespace into which the Model Endpoint can be deployed.

Listing Model Endpoints

```
$ curl -H "Content-Type: application/json" -H "Authorization: Bearer
${CDP_TOKEN}" "https://${DOMAIN}/api/v1alpha1/listEndpoints" -d '{
  "namespace": "serving-default"
}'

{"endpoints":[
  {
    "namespace":"serving-default",
    "name":"mlflow-wine-test-from-registry-onnx",

"url":"https://<domain>/namespaces/serving-default/endpoints/<endpoint_name>/v2/models/<model_name>/infer",
    "state":"Loaded"
  }
]}%
```

Describing a Model Endpoint

To understand the input data shape for a model endpoint, we can follow the Open Inference Protocol V2 Model Metadata API.

```
$ curl -H "Content-Type: application/json" -H "Authorization: Bearer
${CDP_TOKEN}"
"https://${DOMAIN}/namespaces/serving-default/endpoints/mlflow-wine-test-from-registry-onnx/v2/models/model"
```

```
{
  "name":"model",
  "versions":["1"],
  "platform":"onnxruntime_onnx",
  "inputs":[
    {"name":"float_input","datatype":"FP32","shape":[-1,11]}
  ],
  "outputs":[
    {"name":"variable","datatype":"FP32","shape":[-1,1]}
  ]
}%
```

Refer to Triton server documentation for more details on the model input and output specifications.

Making an inference call to a Model Endpoint

Open Inference Protocol

Cloudera AI Inference serves ONNX models using the Nvidia Triton Server. The deployed endpoints are compliant with Open Inference Protocol version 2. For this reason, the model inference request and response payloads are different from models deployed on a CML workspace.

Based on the metadata above, you can construct the input of the inference call based on the Open Inference Protocol V2:

```
# cat ./examples/mlflow/wine-input.json

{
  "parameters": {
    "content_type": "pd"
  },
  "inputs": [
    {
      "name": "float_input",
      "shape": [
        1,
        11
      ],
      "datatype": "FP32",
      "data": [
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4,
        7.4
      ]
    }
  ]
}
```

```
$ curl -H "Content-Type: application/json" -H "Authorization: Bearer
${CDP_TOKEN}"
"https://${DOMAIN}/namespaces/serving-default/endpoints/mlflow-wine-t
est-from-registry-onnx/v2/models/model/infer" -d
@./examples/mlflow/wine-input.json

{
```

```
  "model_name":"model",
  "model_version":"1",
  "outputs":[
    {
      "name":"variable",
      "datatype":"FP32",
      "shape":[1,1],
      "data":[5.535987377166748]
    }
  ]
}
```

OpenAI Inference Protocol

Language models for text generation (TRT-LLM and Transformer) are deployed on NVIDIA's NIM microservices. These endpoints are compliant with the OpenAI Protocol.

An example inference payload for the OpenAI Protocol:

```
{
    "messages": [
        {
            "content": "You are a polite and respectful chatbot
helping people plan a vacation.",
            "role": "system"
        },
        {
            "content": "What should I do for a 4 day vacation in
Spain?",
            "role": "user"
        }
    ],
    "model": "nemo-llama2-7b-chat-from-ngc",
    "max_tokens": 200,
    "top_p": 1,
    "n": 1,
    "stream": false,
    "stop": "\n",
    "frequency_penalty": 0.0
```

```
}
```

Deleting a Model Endpoint

To delete the model endpoint deployed above

```
$ curl -v  -H "Content-Type: application/json" -H "Authorization:
Bearer ${CDP_TOKEN}" "https://${DOMAIN}/api/v1alpha1/deleteEndpoint"
-d '{"namespace": "serving-default", "name":
"mlflow-wine-test-from-registry-onnx"}'
```

Autoscaling Model Endpoints

Model endpoints deployed on Cloudera AI Inference can be configured to auto-scale to 0 instances when there is no load. The following deployment specification shows an example endpoint that will autoscale between 0 and 4 replicas

```
# cat ./examples/mlflow/model-spec-cml-registry.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 1,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  },
  "autoscaling": {
    "min_replicas": "0",
    "max_replicas": "4"
  }
}
```

Tuning Autoscaling Sensitivity

To customize the autoscaling sensitivity and requisites, set the **target** and **metric** fields in the **autoscalingconfig** parameter. The **metric** field is the data we are watching to make autoscaling decisions, which can be set to either **concurrency** or **rps**.

- **Concurrency** is the number of requests that each replica of the model should aim to handle at once.
- **RPS** (requests per second) is the calculated requests per second handled over the polling period.
- **Target** is the target value of the metric that you aim to maintain. If the **Target** value is exceeded when calculating the metric value, new replicas will be spun up or down to maintain the `target` value as the upper bound.

These values can be set as follows:

```
# cat ./examples/mlflow/model-spec-cml-registry.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 1,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  },
  "autoscaling": {
    "min_replicas": "0",
    "max_replicas": "4",
    "autoscalingconfig": {
        "metric": "concurrency",
        "target": "25"
    }
  }
}
```

The example above will scale between zero and four replicas aiming to maintain at most 25 concurrent requests per replica, scaling the number of replicas deployed to maintain this target.

Running Models on GPU

To run a model endpoint on GPU, specify the number of GPUs to use per model replica, as follows. You must specify the GPU count as a *string*.

```
# cat ./examples/mlflow/model-spec-cml-registry.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 1,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  },
  "autoscaling": {
    "min_replicas": "0",
    "max_replicas": "4"
  },
  "resources": {
    "num_gpus": "1"
  }
}
```

The resources field supports specifying the following properties:

1. req_cpu - the requested number of CPU cores, which is a string as per Kubernetes standards.
2. req_memory - the requested amount of memory, which again follows Kubernetes conventions.
3. num_gpus - the requested number of GPUs.

Deploying Models with Canary Deployment

Cloudera AI Inference Service allows users to control traffic percentage to specific model deployments. When initially deploying a model, the traffic value defaults to 100. Following is an example payload:

```
# cat ./examples/mlflow/model-spec-cml-registry.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 1,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  }
}
```

After deploying a model to the endpoint, traffic can be directed to the “canary” model by updating the model with the desired traffic value set:

```
# cat ./examples/mlflow/model-spec-cml-registry-canary.json
{
  "namespace": "serving-default",
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 2,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  }
  "traffic": "35",
}
```

Now, the “traffic” percent of the traffic will be diverted to the newly deployed model version while the remainder is directed to the previously deployed model version.

When you are ready for your canary model to serve all of your incoming requests, you can update the inference endpoint as follows:

```
# cat ./examples/mlflow/model-spec-cml-registry-promote.json
{
  "namespace": "serving-default",
```

```
  "name": "mlflow-wine-test-from-registry-onnx",
  "source": {
    "registry_source": {
      "version": 2,
      "model_id": "yf0o-hrxq-l0xj-8tk9"
    }
  }
  "traffic": "100",
}
```

To reach the model, you can use the following payload:

```
curl -H "Content-Type: application/json" -H "Authorization: Bearer
${CDP_TOKEN}"
https://${DOMAIN}/namespaces/serving-default/endpoints/mlflow-wine-te
st-from-registry-onnx/v2/models/yf0o-hrxq-l0xj-8tk9/infer -d
@./examples/url/wine-input.json
{"model_name":"yf0o-hrxq-l0xj-8tk9","model_version":"1","outputs":[{"
name":"variable","datatype":"FP32","shape":[1,1],"data":[0.0]}]}
```

Note: The model version is always set to 1 which is a known issue.

GUI Client

CML Serving UI provides new features to view, create and edit model endpoints from the control plane UI. Each CML Serving app could have multiple deployed model endpoints. The Model Endpoints page in CDP provides an aggregated list view of all the deployed model endpoints from CML Serving apps. The CML Serving UI is enabled by the ‘ML_ENABLE_ML_SERVING’ entitlement.

CML Serving UI contains following pages:

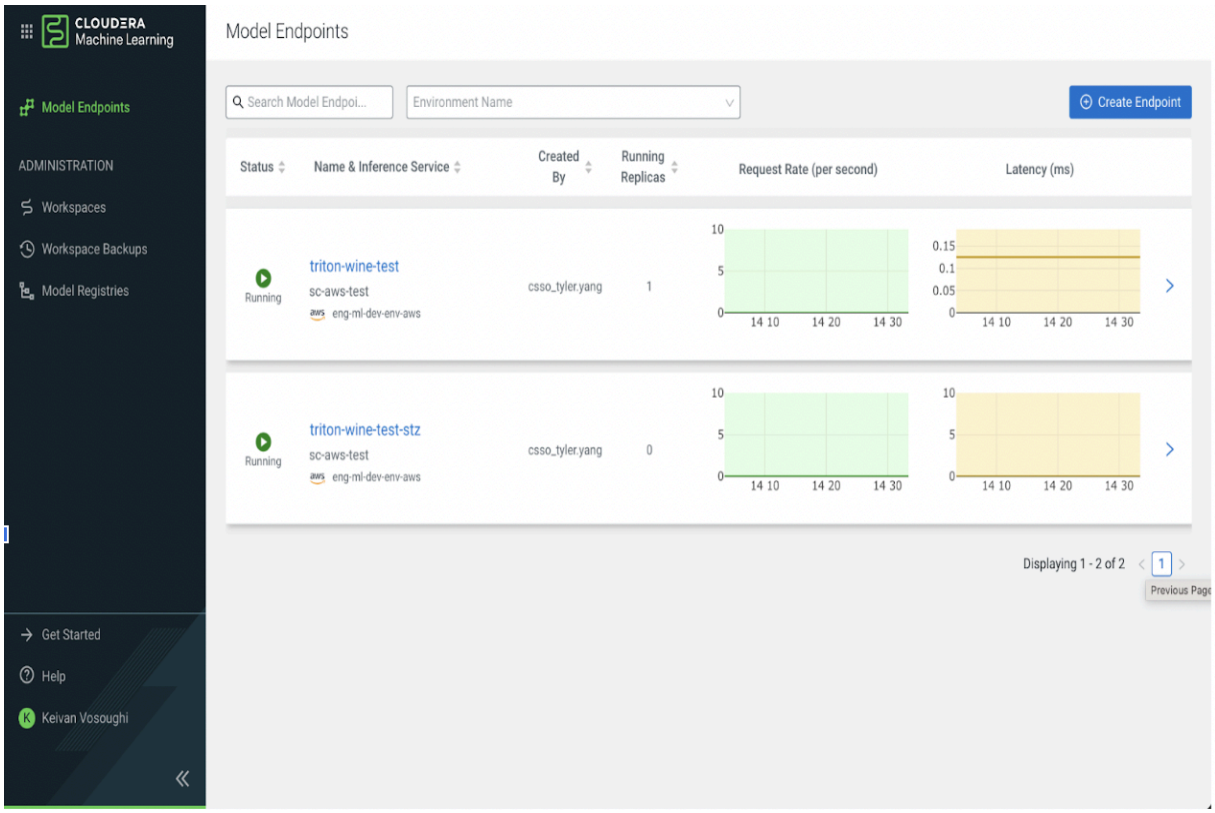
- List Model Endpoints Page
- Create Model Endpoint Page
- Model Endpoint Details Page

Listing Model Endpoints

List Model Endpoints page displays a table view representing an aggregated list of all the model endpoints across the running CML Serving apps.

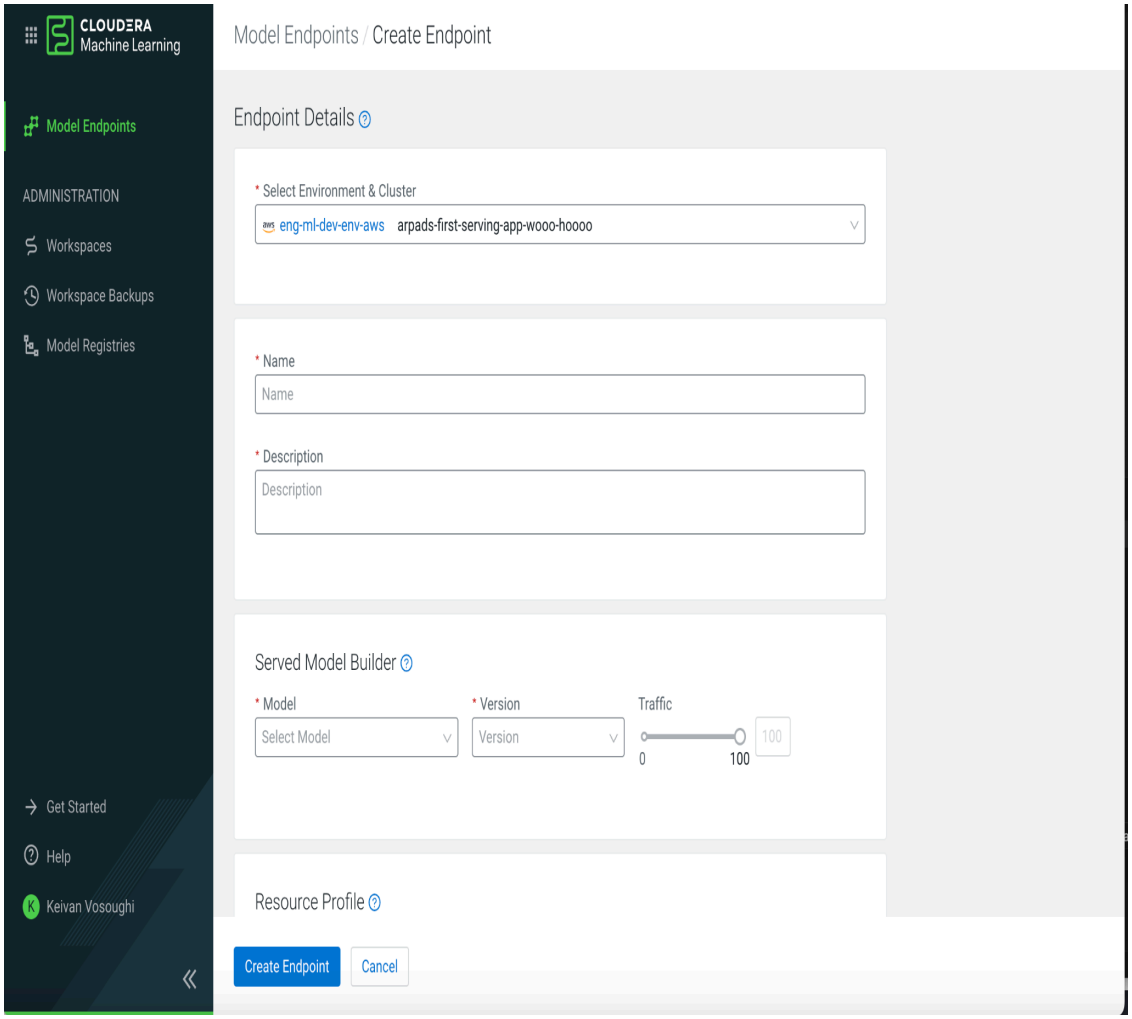
It provides following features:

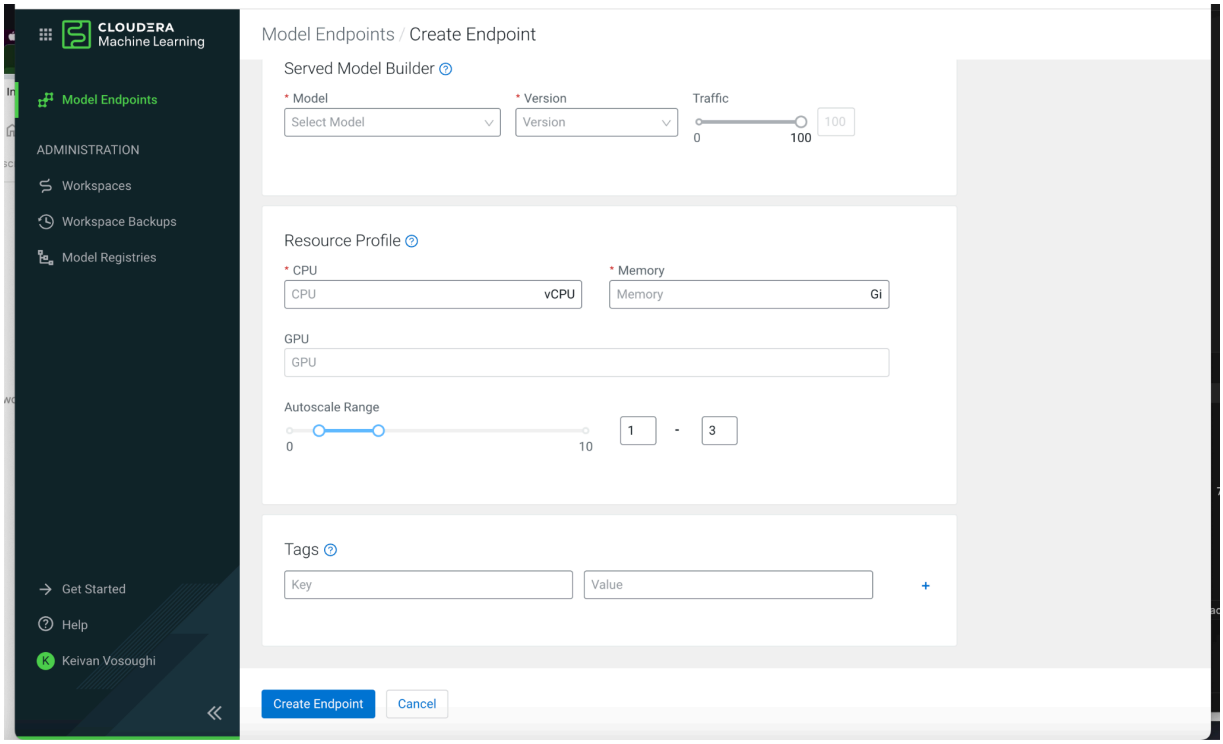
- Filter by Name and Environment
- Model Endpoint Metric Charts
- Table view displays following columns:
    - Status
    - Name & Inference Service
    - Created At
    - Created By
    - Number of replicas



Creating a Model Endpoint

Create Model Endpoint page allows you to select a specific serving app and a deployed model version from Model Registry to create a new model endpoint.



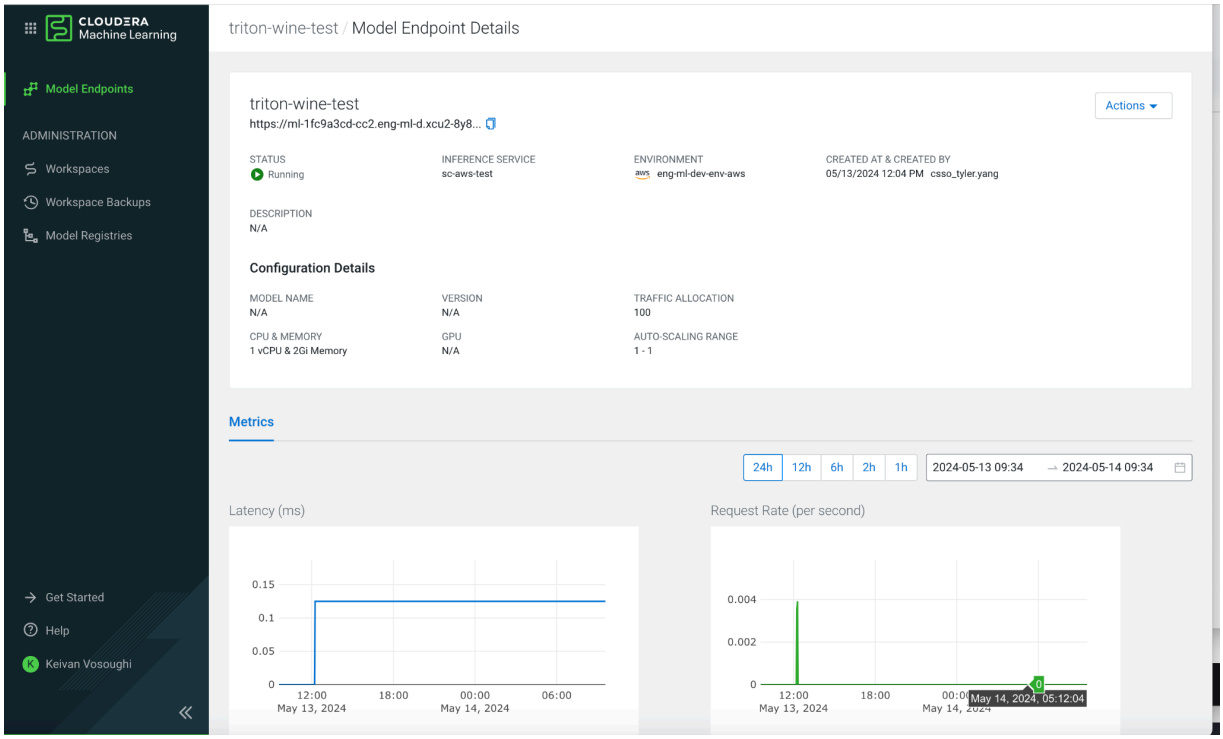


Viewing Model Endpoint Details

Model endpoint details page provides following features:

- Displays Endpoint Details
- Edit Configuration
    - Add Model Version
    - Change Resources Memory/CPU/GPU

- Metric Charts displaying:
    - Request Latency
    - Request Rate
    - CPU Utilization
    - CPU Memory
    - GPU Utilization
    - GPU Memory
    - Replicas
    - Request Error Rate

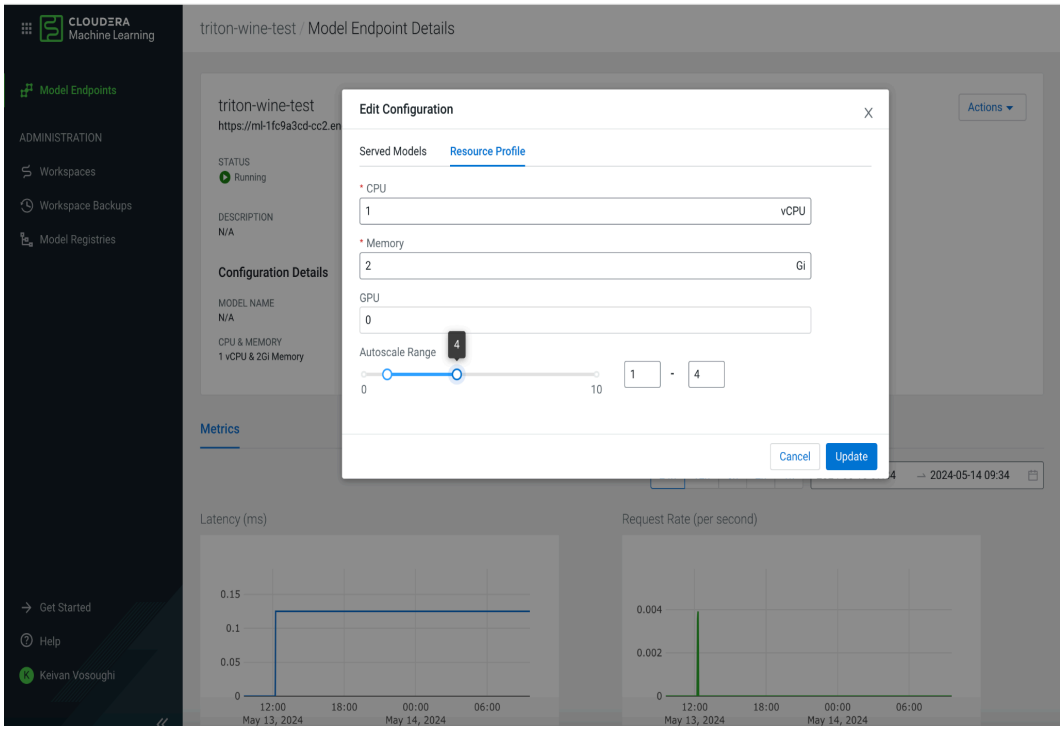


Editing a Model Endpoint Configuration

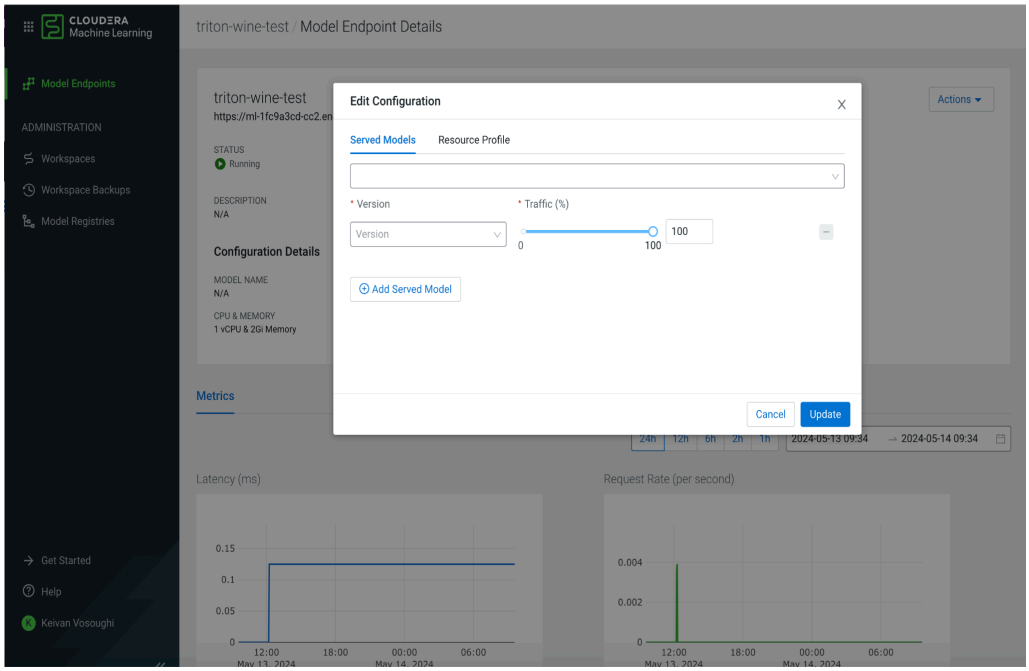
From the Model Endpoint Details page, you can edit the configuration of a specific model endpoint by using the **Actions** menu dropdown.

There are two main tabs for editing the configuration of a model endpoint:

- Resource Profile (Change CPU/GPU/Memory)



- Served Model (Add Model Version)



How Do I

Deploying a TRT-LLM Optimized Large Language Model

You can now deploy a 70b Llama2 TRT-LLM model with chat capability. You must load this model into your model registry. Note, that you need to upgrade the model registry to version 1.4.0-b31 before loading models into it from NGC. Looking at this entry of the supported TRT-LLM models table, you can determine the type and number of GPUs you need to deploy this model. In the table, it mentions 4 A100 80Gb GPUs as a requirement and the corresponding p4d.24xlarge (AWS) and Standard_NC96ads_A100_v4 (Azure) instance types. To load the model into the model registry, first you need to locate the model registry's domain name from the list model registries CDP CLI output:

```
$ cdp ml list-model-registries
```

From the command output, choose the model registry that is in the CDP environment in which you will be creating the Cloudera AI Inference cluster. The registry's domain can be found in the `domain` field.

```
$ MR_DOMAIN=<domain of the registry>
```

To register a model from NGC for example, execute the following query:

```
$ curl -H "Content-Type: application/json" \
-H "Authorization: Bearer ${CDP_TOKEN}" \
"${MR_DOMAIN}/api/v2/models" \
-d '{
  "name": "llama2_70b_chat_from_ngc",
  "createModelVersionRequestPayload": {
    "metadata": {"model_repo_type": "NGC"},
    "downloadModelRepoRequest": {
      "source": "NGC",
      "repo_id":
"ohlfw0olaadg/ea-participants/llama-2-70b-chat:LLAMA-2-70B-CHAT-4K-FP
16-4-A100.24.01"
    }
  }
```

```
}'
```

The result of this request will be something similar to the following:

```
{
  "created_at":"2024-05-07T15:45:54.678Z",
  "creator":{"user_name":"csso_bwiandt"},
  "id":"og6y-qp6f-f69v-4yf3",
  "model_versions":[{
    "created_at":"2024-05-07T15:45:54.682Z",
    "metadata":{"model_repo_type":"NGC","tags":null},
    "model_id":"og6y-qp6f-f69v-4yf3",
    "status":"REGISTERING",
    "tags":null,
    "updated_at":"2024-05-07T15:45:54.682Z",
    "user":{"user_name":"csso_bwiandt"},
    "version":1}],
  "name":"llama2_70b_chat_from_ngc",
  "tags":null,
  "updated_at":"2024-05-07T15:45:54.678Z",
  "visibility":"private"
}
```

From the output you obtained in the **model id**, in this case “`og6y-qp6f-f69v-4yf3`”.

To check the state of the model, run the following request:

```
$ curl -v -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "${MR_DOMAIN}/api/v2/models/og6y-qp6f-f69v-4yf3/versions/1" | jq
```

When the status of the model version becomes READY, it can be deployed to a Cloudera AI Inference service.

To provision a Cloudera AI Inference App first, you have to determine the instance groups you need. The model you are trying to deploy requires 4 A100 GPUs, therefore you have two choices: p4d.24xlarge (AWS) and Standard_NC96ads_A100_v4 (Azure).

The following is the example of Azure and it assembles the payload for the CDP CDLI command:

```
{
    "appName": "testing-a100",
    "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
    "isPrivateCluster": false,
    "provisionK8sRequest": {
        "instanceGroups": [
            {
                "instanceType": "Standard_D4s_v3",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 1,
                    "enabled": true
                }
            },
            {
                "instanceType": "Standard_NC96ads_A100_v4",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 1,
                    "enabled": true
                }
            }
        ],
        "environmentCrn":
"crn:cdp:environments:<YOUR_ENVIRONMENT_CRN>",
        "tags": [
            {
                "key": "experienceType",
```

```
                "value": "cml-serving"
            }
        ]
    },
    "mlservingVersion": "1.1.0-b53",
    "usePublicLoadBalancer": false
}
```

Note: Here, you are setting the GPU node group to autoscale to 0 instances when there's no load to save cloud costs. It is recommended to set the "rootVolume" size to 1Ti as the large language model artifacts can be hundreds of gigabytes large when uncompressed. With this payload saved to a local file, you can start creating our cluster:

```
$ cdp ml create-ml-serving-app \
  --cli-input-json file:///tmp/create-serving-app-input.json
```

The response contains the app-crn that you will use in following to view the status of the deployment.

```
$ cdp ml describe-ml-serving-app --app-crn <YOUR_APP_CRN>
```

```
{
    "app": {
        "appName": "<YOUR_APP_NAME>",
        "appCrn": "<YOUR_APP_CRN>",
        "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
        "environmentName": "<YOUR_ENVIRONMENT_NAME>",
        "ownerEmail": "user@org.com",
        "mlServingVersion": "1.1.0-b53",
        "isPrivateCluster": false,
        "creationDate": "2024-05-07T14:27:16.817000+00:00",
        "cluster": {
            "clusterName": "ml-1fcaa8cf-a94",
            "domainName": "<YOUR_DOMAIN_NAME>",
            "liftieID": "liftie-3544nrdr",
```

```
            "isPublic": false,
            "ipAllowlist": "0.0.0.0/0",
            "authorizedIpRangesAllowList": false
        },
        "status": "installation:finished",
        "usePublicLoadBalancer": false,
        "httpsEnabled": true
    }
}
```

When the status in the above response becomes *installation:finished*, you can start the deployment of the previously registered model. First you need to obtain the domain name of the Cloudera AI Inference App. The domain is part of the output above.

```
$ export DOMAIN=<YOUR_DOMAIN_NAME>
```

To deploy our model, the first step is to issue a *deployEndpoint* call to our Cloudera AI Inference App's API:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/deployEndpoint" -d \
  '{
    "namespace": "serving-default",
    "name": "llama2-70b-from-ngc",
    "source": {
      "registry_source": {
        "version": 1,
        "model_id": "og6y-qp6f-f69v-4yf3"
      }
    },
    "resources": {
      "num_gpus": "4",
      "req_cpu": "10",
      "req_memory": "512Gi"
    }
```

```
  }'
```

You can follow the status of the endpoint through the describe endpoint:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/describeEndpoint" -d \
  '{
    "namespace": "serving-default",
    "name": "llama2-70b-from-ngc"
  }'
```

```
{
  "namespace": "serving-default",
  "name": "llama2-70b-from-ngc",
  "url": "",
  "conditions": [...]
  "status": {
    "failed_copies": 0,
    "total_copies": 0,
    "active_model_state": "",
    "target_model_state": "Loading",
    "transition_status": "InProgress"
  },
  "observed_generation": 2,
  "replica_count": 0,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-07T18:06:21Z",
  "resources": {
    "req_cpu": "10",
    "req_memory": "512Gi",
    "num_gpus": "4"
  },
  "source": {
    "registry_source": {
```

```
      "model_id": "og6y-qp6f-f69v-4yf3",
      "version": 1
    }
  },
  "autoscaling": {...},
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "og6y-qp6f-f69v-4yf3",
        "version": 1
      }
    },
    "previous_model": null
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
  "api_standard": "openai",
  "chat": true
}
```

From this output, you can infer the following information:

- Our model comes from the CML Registry, its id is `og6y-qp6f-f69v-4yf3`, version is `1`.
- The model endpoint is conforming to the OpenAI API standard.
- The “`chat`” attribute is true, therefore the model is able to chat and responds to both */v1/completions* and */v1/chat/completions* endpoints.
- Loading is in progress, URL is not yet present for this endpoint.

With the model fully loaded, the output will look similar to the following:

```
{
  "namespace": "serving-default",
  "name": "llama2-70b-from-ngc",
  "url":
"https://ml-1fcaa8cf-a94.eng-ml-i.svbr-nqvp.int.cldr.work/namespaces/
serving-default/endpoints/llama2-70b-from-ngc/v1/chat/completions",
```

```
 "conditions": [
   {
     "type": "IngressReady",
     "status": "True",
     "severity": "",
     "last_transition_time": "1715111857",
     "reason": "",
     "message": ""
   },
   {
     "type": "LatestDeploymentReady",
     "status": "True",
     "severity": "Info",
     "last_transition_time": "1715111857",
     "reason": "",
     "message": ""
   },
   {
     "type": "PredictorConfigurationReady",
     "status": "True",
     "severity": "Info",
     "last_transition_time": "1715111857",
     "reason": "",
     "message": ""
   },
   {
     "type": "PredictorReady",
     "status": "True",
     "severity": "",
     "last_transition_time": "1715111857",
     "reason": "",
     "message": ""
   },
   {
     "type": "PredictorRouteReady",
     "status": "True",
     "severity": "Info",
     "last_transition_time": "1715111857",
     "reason": "",
```

```
      "message": ""
    },
    {
      "type": "Ready",
      "status": "True",
      "severity": "",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "RoutesReady",
      "status": "True",
      "severity": "Info",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    }
  ],
  "status": {
    "failed_copies": 0,
    "total_copies": 1,
    "active_model_state": "Loaded",
    "target_model_state": "Loaded",
    "transition_status": "UpToDate"
  },
  "observed_generation": 2,
  "replica_count": 1,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-07T18:06:21Z",
  "resources": {
    "req_cpu": "10",
    "req_memory": "512Gi",
    "num_gpus": "4"
  },
  "source": {
    "registry_source": {
      "model_id": "og6y-qp6f-f69v-4yf3",
```

```
      "version": 1
    }
  },
  "autoscaling": {
    "min_replicas": "1",
    "max_replicas": "1",
    "autoscalingconfig": {
      "metric": "",
      "target": "",
      "target_utilization": "",
      "scale_to_zero_retention": "",
      "activation_scale": "",
      "scale_down_delay": "",
      "panic_window_percentage": "",
      "panic_threshold": "",
      "stable_window": ""
    }
  },
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "og6y-qp6f-f69v-4yf3",
        "version": 1
      }
    },
    "previous_model": null
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
  "api_standard": "openai",
  "chat": true
}
```


Note that it can take up to 50 minutes for a large language model of this size to be downloaded and initialized. After the “active_model_state” of the model endpoint is “Loaded”, the model is ready to accept inference requests.

To interact with this model, you must first obtain the identifier of the model that is loaded by the runtime. For this, issue a GET request to the `/v1/models` path of the model endpoint:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}"
"${DOMAIN}/namespaces/serving-default/endpoints/<ENDPOINT_NAME>/v1/mo
dels | jq
```

You can notice that the model identifier is formed from the `model id` from the registry and the `model version`:

```
{
  "object": "list",
  "data": [
    {
      "id": "ax30-jx9a-qqlc-f2cc",
      "object": "model",
      "created": 1715710806,
      "owned_by": "nim_llm_backend",
      "root": "ax30-jx9a-qqlc-f2cc",
      "parent": null,
      "permission": [
        {
          "id": "modelperm-027d8ad3-22f7-4827-84ff-938511d80388",
          "object": "model_permission",
          "created": 1715710806,
          "allow_create_engine": false,
          "allow_sampling": true,
          "allow_logprobs": true,
          "allow_search_indices": false,
          "allow_view": true,
          "allow_fine_tuning": false,
          "allow_chat": true,
          "organization": "*",
          "group": null,
          "is_blocking": false
        }
      ]
```

```
    }
  ]
}
```

You now have the inference endpoint and the model identifier, you can issue the following inference request:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
"${DOMAIN}/namespaces/serving-default/endpoints/llama2-70b-from-ngc/v
1/chat/completions -d '{
    "messages": [
        {
            "content": "You are a polite and respectful chatbot
helping people plan a vacation.",
            "role": "system"
        },
        {
            "content": "What should I do for a 4 day vacation in
Spain?",
            "role": "user"
        }
    ],
    "model": "ax30-jx9a-qqlc-f2cc",
    "max_tokens": 200,
    "top_p": 1,
    "n": 1,
    "stream": false,
    "frequency_penalty": 0.0
}'
```

Deploying a Hugging Face Transformer Large Language Model

You can now deploy a 70b Llama2 AWQ quantized Transformer model from Huggingface. You must first load this model into your model registry. The model chosen for this demonstration is the TheBloke/Llama-2-70B-Chat-AWQ. If you see the tokenizer config of this model, you can

observe that it does not have a chat template. This will result in our model responding only on the *`/v1/completions`* api endpoint. Note, that you need to upgrade the model registry to version 1.4.0-b31 before you load models into it from Huggingface. The 70b Llama2 AWQ model can be loaded on 2A10 GPUs. For example, on AWS, you can provision our cluster with the g5.12xlarge instance type, since it has 4 A10 GPUs. To load the model into the model registry, first you have to locate the model registry's domain name from the list model registries CDP CLI output:

```
$ cdp ml list-model-registries
```

From the output, choose the model registry that is in the CDP environment in which you will be creating the Cloudera AI Inference cluster. The registry's domain can be found in the `domain` field.

```
$ MR_DOMAIN=<domain of the registry>
```

To register a model from Hugging Face, for example, execute the following query:

```
$ curl -H "Content-Type: application/json" \
-H "Authorization: Bearer ${CDP_TOKEN}" \
"${MR_DOMAIN}/api/v2/models" \
-d '{
  "name": "llama2_70b_chat_from_hf",
  "createModelVersionRequestPayload": {
    "metadata": {"model_repo_type": "HF"},
    "downloadModelRepoRequest": {
      "source": "HF",
      "repo_id": "TheBloke/Llama-2-70B-Chat-AWQ"
    }
  }
}'
```

The result of this request will be similar to the following:

```
{
  "created_at":"2024-05-07T15:45:54.678Z",
```

```
  "creator":{"user_name":"csso_user"},
  "id":"og6y-qp6f-f69v-4yf3",
  "model_versions":[{
    "created_at":"2024-05-07T15:45:54.682Z",
    "metadata":{"model_repo_type":"HF","tags":null},
    "model_id":"og6y-qp6f-f69v-4yf3",
    "status":"REGISTERING",
    "tags":null,
    "updated_at":"2024-05-07T15:45:54.682Z",
    "user":{"user_name":"csso_user"},
    "version":1}],
  "name":"llama2_70b_chat_from_hf",
  "tags":null,
  "updated_at":"2024-05-07T15:45:54.678Z",
  "visibility":"private"
}
```

From the output you can obtain the model id, in this case “`og6y-qp6f-f69v-4yf3`”.

To check the state of the model, issue the following request:

```
$ curl -v -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "${MR_DOMAIN}/api/v2/models/og6y-qp6f-f69v-4yf3/versions/1" | jq
```

When the status of the model version becomes `READY`, it can be deployed to a Cloudera AI Inference service.

To provision a Cloudera AI Inference App, first you have to determine the instance groups you need. The model you are trying to deploy requires 2 A10 GPUs. Therefore you have two choices: g5.12xlarge (AWS) and Standard_NV72ads_A10_v5 (Azure). We have chosen AWS and assemble the payload for the *`cdp cli`* command:

```
{
    "appName": "testing-a10",
    "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
    "isPrivateCluster": false,
```

```
    "provisionK8sRequest": {
        "instanceGroups": [
            {
                "instanceType": "m5.4xlarge",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 1,
                    "enabled": true
                }
            },
            {
                "instanceType": "g5.12xlarge",
                "instanceCount": 1,
                "rootVolume": {
                    "size": 1024
                },
                "autoscaling": {
                    "minInstances": 0,
                    "maxInstances": 1,
                    "enabled": true
                }
            }
        ],
        "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
        "tags": [
            {
                "key": "experienceType",
                "value": "cml-serving"
            }
        ]
    },
    "mlservingVersion": "1.1.0-b53",
    "usePublicLoadBalancer": false
}
```

Note that you are setting the GPU node group to autoscale to 0 instances when there is no load to save cloud costs. It is recommended to set the “rootVolume” size to 1Ti (1024Gi) since large language model artifacts can be hundreds of gigabytes large when uncompressed. With this payload saved to a local file, you can start creating the cluster:

```
$ cdp ml create-ml-serving-app \
  --cli-input-json file:///tmp/create-serving-app-input.json
```

The response contains the app-crn that you will use in following the status of the deployment.

```
$ cdp ml describe-ml-serving-app --app-crn <YOUR_APP_CRN>
```

```
{
    "app": {
        "appName": "<YOUR_APP_NAME>",
        "appCrn": "<YOUR_APP_CRN>",
        "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
        "environmentName": "<YOUR_ENVIRONMENT_NAME>",
        "ownerEmail": "user@org.com",
        "mlServingVersion": "1.1.0-b53",
        "isPrivateCluster": false,
        "creationDate": "2024-05-07T14:27:16.817000+00:00",
        "cluster": {
            "clusterName": "ml-1fcaa8cf-a94",
            "domainName": "<YOUR_DOMAIN_NAME>",
            "liftieID": "liftie-3544nrdr",
            "isPublic": false,
            "ipAllowlist": "0.0.0.0/0",
            "authorizedIpRangesAllowList": false
        },
        "status": "installation:finished",
        "usePublicLoadBalancer": false,
        "httpsEnabled": true
    }
}
```

When the status in the above response becomes *installation:finished*, you can start deployment of the previously registered model. To do that, you have to obtain the domain name of the Cloudera AI Inference App. The domain is part of the output above.

```
$ export DOMAIN=<YOUR_DOMAIN_NAME>
```

To deploy the model, you must issue a deployEndpoint call to our Cloudera AI Inference App's API:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/deployEndpoint" -d \
'{
    "namespace": "serving-default",
    "name": "llama2-70b-from-hf",
    "source": {
      "registry_source": {
        "version": 1,
        "model_id": "og6y-qp6f-f69v-4yf3"
      }
    },
    "resources": {
      "num_gpus": "2",
      "req_cpu": "10",
      "req_memory": "64Gi"
    }
}'
```

You can follow the status of the endpoint through the describe endpoint:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/describeEndpoint" -d \
'{
    "namespace": "serving-default",
    "name": "llama2-70b-from-hf"
}'
```

```
{
  "namespace": "serving-default",
  "name": "llama2-70b-from-hf",
  "url": "",
  "conditions": [...]
  "status": {
    "failed_copies": 0,
    "total_copies": 0,
    "active_model_state": "",
    "target_model_state": "Loading",
    "transition_status": "InProgress"
  },
  "observed_generation": 2,
  "replica_count": 0,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-07T18:06:21Z",
  "resources": {
    "req_cpu": "10",
    "req_memory": "64Gi",
    "num_gpus": "2"
  },
  "source": {
    "registry_source": {
      "model_id": "og6y-qp6f-f69v-4yf3",
      "version": 1
    }
  },
  "autoscaling": {...},
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "og6y-qp6f-f69v-4yf3",
        "version": 1
      }
    },
    "previous_model": null
```

```
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
  "api_standard": "openai",
  "chat": false
}
```

From this output we can deduce the following information:

- Our model comes from the CML Registry, its id is `og6y-qp6f-f69v-4yf3`, version is `1`.
- The model endpoint is conforming to the OpenAI API standard.
- "`chat`" attribute is false, therefore the model is only able to respond on the */v1/completions* endpoint
- Loading is in progress, url is not yet present for this endpoint

With the model fully loaded, the output will look like this:

```
{
  "namespace": "serving-default",
  "name": "llama2-70b-from-hf",
  "url":
"https://ml-1fcaa8cf-a94.eng-ml-i.svbr-nqvp.int.cldr.work/namespaces/
serving-default/endpoints/llama2-70b-from-hf/v1/completions",
  "conditions": [
    {
      "type": "IngressReady",
      "status": "True",
      "severity": "",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "LatestDeploymentReady",
      "status": "True",
      "severity": "Info",
```

```
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "PredictorConfigurationReady",
      "status": "True",
      "severity": "Info",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "PredictorReady",
      "status": "True",
      "severity": "",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "PredictorRouteReady",
      "status": "True",
      "severity": "Info",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "Ready",
      "status": "True",
      "severity": "",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    },
    {
      "type": "RoutesReady",
      "status": "True",
```

```
      "severity": "Info",
      "last_transition_time": "1715111857",
      "reason": "",
      "message": ""
    }
  ],
  "status": {
    "failed_copies": 0,
    "total_copies": 1,
    "active_model_state": "Loaded",
    "target_model_state": "Loaded",
    "transition_status": "UpToDate"
  },
  "observed_generation": 2,
  "replica_count": 1,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-07T18:06:21Z",
  "resources": {
    "req_cpu": "10",
    "req_memory": "64Gi",
    "num_gpus": "2"
  },
  "source": {
    "registry_source": {
      "model_id": "og6y-qp6f-f69v-4yf3",
      "version": 1
    }
  },
  "autoscaling": {
    "min_replicas": "1",
    "max_replicas": "1",
    "autoscalingconfig": {
      "metric": "",
      "target": "",
      "target_utilization": "",
      "scale_to_zero_retention": "",
      "activation_scale": "",
      "scale_down_delay": "",
```

```
      "panic_window_percentage": "",
      "panic_threshold": "",
      "stable_window": ""
    }
  },
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "og6y-qp6f-f69v-4yf3",
        "version": 1
      }
    },
    "previous_model": null
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
  "api_standard": "openai",
  "chat": false
}
```

Note that it can take up to 20-30 minutes for a large language model of this size to be downloaded and initialized. Once the “active_model_state” of the model endpoint is “Loaded”, the model is ready to accept inference requests.

To interact with this model, you have to first obtain the identifier of the model that is loaded by the runtime. For this, issue a GET request to the `/v1/models` path of the model endpoint:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}"
"${DOMAIN}/namespaces/serving-default/endpoints/<ENDPOINT_NAME>/v1/mo
dels | jq
```

You can notice that the model identifier is formed from the **model id** from the registry and the **model version**:

```
{
  "object": "list",
  "data": [
    {
      "id": "ax30-jx9a-qqlc-f2cc",
      "object": "model",
      "created": 1715710806,
      "owned_by": "nim_llm_backend",
      "root": "ax30-jx9a-qqlc-f2cc",
      "parent": null,
      "permission": [
        {
          "id": "modelperm-027d8ad3-22f7-4827-84ff-938511d80388",
          "object": "model_permission",
          "created": 1715710806,
          "allow_create_engine": false,
          "allow_sampling": true,
          "allow_logprobs": true,
          "allow_search_indices": false,
          "allow_view": true,
          "allow_fine_tuning": false,
          "allow_chat": false,
          "organization": "*",
          "group": null,
          "is_blocking": false
        }
      ]
    }
  ]
}
```

Now you know the inference endpoint and the model identifier, you can issue the following inference request:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
"${DOMAIN}/namespaces/serving-default/endpoints/llama2-70b-from-hf/v1
```

```
/completions -d '{
    "prompt": "What should I do for a 4 day vacation in Spain?",
    "model": "ax30-jx9a-qqlc-f2cc",
    "max_tokens": 200,
    "top_p": 1,
    "n": 1,
    "stream": false,
    "frequency_penalty": 0.0
}'
```

Deploying a Predictive Deep Learning Model

In this example, you can see how to deploy the resnet-18 image classification model from ONNX Model Zoo (here). The first step is to import the model into a session in CML Workspace, which must be in the same CDP environment as the AI Inference service and CML Registry. To do this, perform the following steps:

1. Upload the model artifact to your project's file system
    a. From the Project Overview page, upload from your local computer using the "upload" button"

In your session, you can then load the artifact using the following script and load it to the CML experiments page as follows:

```
!pip install onnx mlflow onnxruntime

import onnx
import mlflow
import onnxruntime

resnet18 = onnx.load("resnet18-v1-7.onnx")
mlflow.set_experiment("resnet18")
with mlflow.start_run() as run:
    mlflow.onnx.log_model(resnet18,
                          "resnet-demo",
                          registered_model_name="model1")
```

Navigate to the experiments page and locate your experiment. Select the desired experiment run and select the artifact folder. You can then select “Register model to upload to model registry”

After doing this, you can navigate to our AI Inference Service to deploy the model using the following payload to deploy the resnet18 model into our serving cluster. Replace the model_id and model_version with the matching values from the workspace model registry page:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/deployEndpoint" -d \
'{
    "namespace": "serving-default",
    "name": "resnet-18-onnx",
    "source": {
      "registry_source": {
        "version": 1,
        "model_id": "pr5z-mc4s-hrxq-5zg4"
      }
    }
}
```

You can follow the status of this model through the describe model endpoint:

```
$ curl -H "Content-Type: application/json" \
  -H "Authorization: Bearer ${CDP_TOKEN}" \
  "https://${DOMAIN}/api/v1alpha1/describeEndpoint" -d \
'{
    "namespace": "serving-default",
    "name": "resnet18-onnx"
}'
```

Which should return the following response:

```
{
  "namespace": "serving-default",
  "name": "resnet18-onnx",
```

```
  "url": "",
  "conditions": [...]
  "status": {
    "failed_copies": 0,
    "total_copies": 0,
    "active_model_state": "",
    "target_model_state": "Loading",
    "transition_status": "InProgress"
  },
  "observed_generation": 2,
  "replica_count": 0,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-10T19:22:21Z",
  "resources": {
    "req_cpu": "1",
    "req_memory": "2Gi"
    "num_gpus": "N/A"
  },
  "source": {
    "registry_source": {
      "model_id": "pr5z-mc4s-hrxq-5zg4",
      "version": 1
    }
  },
  "autoscaling": {...},
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "pr5z-mc4s-hrxq-5zg4",
        "version": 1
      }
    },
    "previous_model": null
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
```

```
  "api_standard": "oip",
  "chat": false
}
```

From this output you can infer the following information:

- Our model comes from the CML Registry, its id is `pr5z-mc4s-hrxq-5zg4`, version is `1`.
- The model endpoint is conforming to the Open Inference Protocol (OIP) API standard.
- "`chat`" attribute is false, therefore the model is only able to respond on the */v2/models/<model_name>/infer* endpoint

Loading is in progress, URL is not yet present for this endpoint

Eventually, the model will become ready and the describe payload should appear similar to the following:

```
{
  "namespace": "serving-default",
  "name": "resnet18-onnx",
  "url":
"https://ml-1fcaa8cf-a94.eng-ml-i.svbr-nqvp.int.cldr.work/namespaces/
serving-default/endpoints/resnet18-onnx/v2/models/pr5z-mc4s-hrxq-5zg4
/infer",
  "conditions": [ ...
  ],
  "status": {
    "failed_copies": 0,
    "total_copies": 1,
    "active_model_state": "Loaded",
    "target_model_state": "Loaded",
    "transition_status": "UpToDate"
  },
  "observed_generation": 2,
  "replica_count": 1,
  "created_by": "csso_user",
  "description": "",
  "created_at": "2024-05-10T19:22:21Z",
  "resources": {
```

```
    "req_cpu": "1",
    "req_memory": "2Gi",
    "num_gpus": "N/A"
  },
  "source": {
    "registry_source": {
      "model_id": "pr5z-mc4s-hrxq-5zg4",
      "version": 1
    }
  },
  "autoscaling": {
    "min_replicas": "1",
    "max_replicas": "1",
    "autoscalingconfig": {
      "metric": "",
      "target": "",
      "target_utilization": "",
      "scale_to_zero_retention": "",
      "activation_scale": "",
      "scale_down_delay": "",
      "panic_window_percentage": "",
      "panic_threshold": "",
      "stable_window": ""
    }
  },
  "endpointmetadata": {
    "current_model": {
      "registry_source": {
        "model_id": "pr5z-mc4s-hrxq-5zg4",
        "version": 1
      }
    },
    "previous_model": null
  },
  "traffic": {
    "current_revision_traffic": "100",
    "previous_revision_traffic": "0"
  },
  "api_standard": "oip",
```

```
  "chat": false
}
```

Now you have loaded the resnet18 model into your serving cluster.

Obtaining the kubeconfig for my Cloudera AI Inference Service Cluster

AWS

As part of this process, you must enter the user's Amazon Resource Name (ARN). Make sure you have access to this information before you begin. Either get the ARN from the user or look up a user's ARN in your AWS account. To obtain a user's ARN from the AWS account, go to your organization's AWS Account > Identity and Access Management (IAM) > Users and lookup the user. The ARN is available on their Summary page.

If you are using the AWS CLI, you can run the following command to get the ARN:

```
$ aws sts get-caller-identity
```

```
# Sample output
{
"UserId": "ABCDE12345FGHIJKLMNO6789",
"Account": "888888888888",
"Arn": "arn:aws:iam::888888888888:user/<username>"
}
```

To obtain your Cloudera AI Inference app CRN, run the following command:

```
$ cdp ml list-ml-serving-apps
```

To grant access to the Cloudera AI Inference AWS cluster, run the following command:

```
$ cdp ml grant-ml-serving-app-access \
  --resource-crn <YOUR_CLOUDERA_AI_INFERENCE_APP_CRN> \
```

```
  --identifier <YOUR_AWS_ARN>
```

You can obtain the kubeconfig used to access the above cluster by running:

```
$ cdp ml get-ml-serving-app-kubeconfig \
  --app-crn <YOUR_CLOUDERA_AI_INFERENCE_APP_CRN> \
  | jq '.kubeconfig' | yq > myconfig.yaml
```

Send the downloaded Kubernetes config file to the user who has been granted access. To connect to the EKS cluster, you must have **aws-iam-authenticator** installed.

Getting the logs of the Cloudera AI Inference API server:

```
$ KUBECONFIG=myconfig.yaml kubectl get logs deployment/api -n serving
```

Azure

Locate the liftie cluster identifier for your Cloudera AI Inference app by issuing a describe command using CDP CLI:

```
$ cdp ml describe-ml-serving-app --app-crn <YOUR_APP_CRN>
```

```
{
    "app": {
        "appName": "<YOUR_APP_NAME>",
        "appCrn": "<YOUR_APP_CRN>",
        "environmentCrn": "<YOUR_ENVIRONMENT_CRN>",
        "environmentName": "<YOUR_ENVIRONMENT_NAME>",
        "ownerEmail": "user@org.com",
        "mlServingVersion": "1.1.0-b53",
        "isPrivateCluster": false,
        "creationDate": "2024-05-07T14:27:16.817000+00:00",
        "cluster": {
            "clusterName": "ml-1fcaa8cf-a94",
            "domainName": "<YOUR_DOMAIN_NAME>",
```

```
            "liftieID": "liftie-3544nrdr",
            "isPublic": false,
            "ipAllowlist": "0.0.0.0/0",
            "authorizedIpRangesAllowList": false
        },
        "status": "installation:finished",
        "usePublicLoadBalancer": false,
        "httpsEnabled": true
    }
}
```

The cluster identifier is in app.cluster.liftieID. Locate this entry in the Azure Portal's Kubernetes services page and do the following:

1. Open the entry.
2. Click on "Connect".
3. Copy the command under "Download cluster credentials" to your terminal and execute.

You are now authenticated and your kubectl context is set up to interact with the AKS cluster.

Known issues and limitations

- When creating the Cloudera AI Inference service cluster on AWS, you must specify a CPU node group before specifying the GPU node group. A GPU-only node group configuration is not supported in this release. Refer to Limitations on AWS for more details.
- Transformer models from Hugging Face will not work on Azure A10 nodes, due to CUDA version mismatch between the NIM LLM container and the Azure VM.
- Inference models will display model version 1 regardless of the model version in the CML Registry for ONNX models.
- Updating description after a model has been added to an endpoint will lead to a UI mismatch in the model builder for models listed by the model builder and the models deployed.
- CORS issue. When using console.altus.cloudera.com logon to the CDP control plane, you might see the CORS issue in the Cloudera AI Inference GUI, use console.cdp.cloudera.com.
- The "cdp ml modify-ml-serving-app" command has a known issue and will not work in this release.

- Model endpoints can only be deployed in the serving-default namespace in the current release.

Manually Updating Model Registry Configuration

If you upgrade the CML Registry *after* creating your Cloudera AI Inference service cluster, the model registry configuration stored by the AI Inference service will get out of sync. Follow the steps below to manually reconcile the configuration, so that AI Inference service will be able to connect to the model registry.

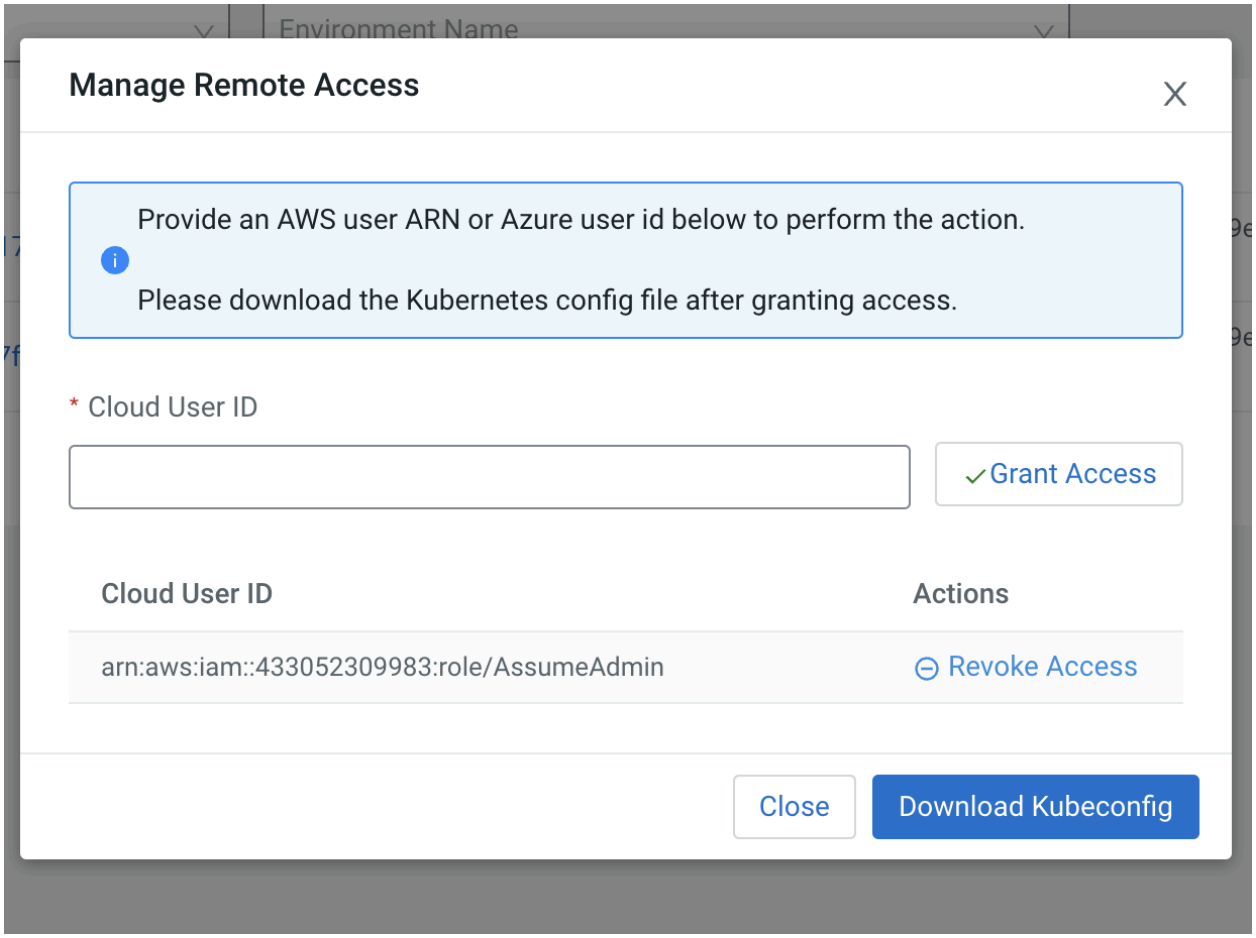
Note: You are required to access multiple clusters when performing the below steps. For information configuring and switching Kubeconfig's between multiple clusters, see Configure Access to Multiple Clusters

1. Get KubeConfig for AI Inference Service Cluster

Refer to Obtaining the kubeconfig for my Cloudera AI Inference Service Cluster section to obtain your kubeconfig file.

2. Get New ConfigMap Data

1. Find the new model registry ConfigMap. Use the *grant-model-registry-access* API in CDP CLI to add your user name to the new model registry, or use the UI, as seen here:



2. *Once* your user ARN has been granted access to the Model Registry, get the ConfigMap data in the following way:
    - Download the Model Registry Kubeconfig
    - Set the Model Registry Kubeconfig (can save it to **~/.kube/config)**
    - **kubectl get cm -n mlx**
        - Lists all the available ConfigMaps
    - **kubectl describe cm jwks-rootca -n mlx**
        - Returns the TLS Certificate for Model Registry
    - **cdp ml list-model-registries**
        - The response will contain the domain for the updated Model Registry

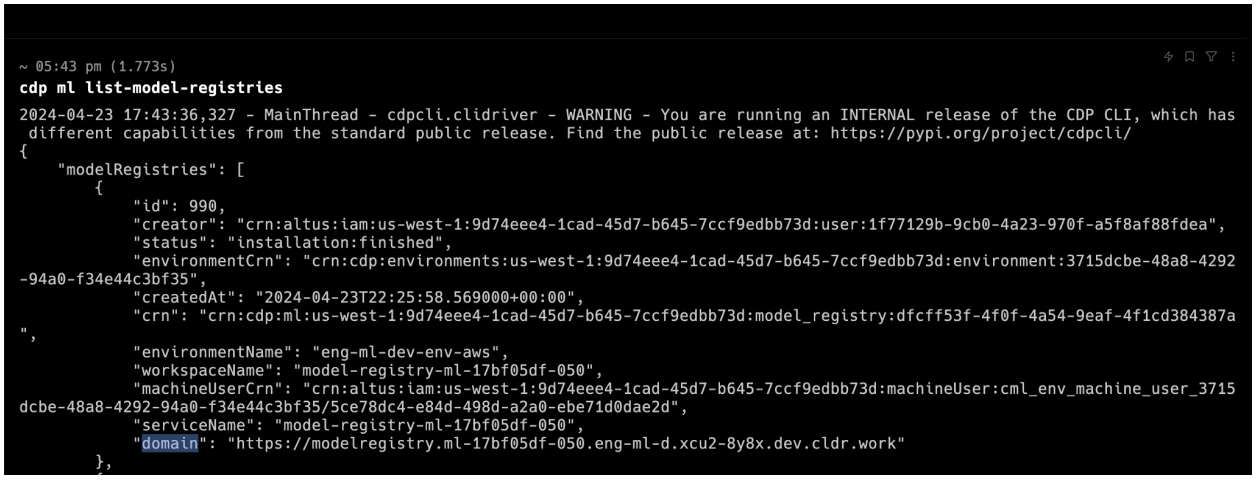
    Make sure to copy the entire rootca.pem output from the command:

    kubectl describe cm jwks-rootca -n mlx

```
              meta.helm.sh/release-namespace: mtx

Data
====
rootca.pem:
----
-----BEGIN CERTIFICATE-----
MIIFmDCCA4CgAwIBAgIQU9C87nMpOIFKYpfvOHFHFDANBgkqhkiG9w0BAQsFADBm
MQswCQYDVQQGEwJVUzEzMDEGA1UEChMqKFNUQUdJTkcpIEludGVybmV0IFNlY3Vy
aXR5IFJlc2VhcmNoIEdyb3VwMSIwIAYDVQQDExkoU1RBR0lORykgUHJldGVuZCBQ
ZWFyIFgxMB4XDTE1MDYwNDExMDQzOFoXDTM1MDYwNDExMDQzOFowZjELMAkGA1UE
BhMCVVMxMzAxBgNVBAoTKihTVEFHSU5HKSBJbnRlcm5ldCBTZWN1cml0eSBSZXNl
YXJjaCBHcm91cDEiMCAGA1UEAxMZKFNUQUdJTkcpIFByZXRlbmQgUGVhciBYMTCC
AiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBALbagEdDTa1QgGBWSYkyMhsc
ZXENOBaVRTMX1hceJENgsL0Ma49D3MilI4KS38mtkmdF6cPWnL++fgehT0FbRHZg
jOEr8UAN4jH6omjrbTD++VZneTsMVaGamQmDdFl5g1gYaigkkmx8OiCO68a4QXg4
wSyn6iDipKP8utsE+x1E28SA75HOYqpdrk4HGxuULvlr03wZGTIf/oRt2/c+dYmD
oaJhge+GOrLAEQByO7+8+vzOwpNAPEx6LW+crEEZ7eBXih6VP19sTGy3yfqK5tPt
TdXXCOQMKAp+gCj/VByhmIr+0iNDC540gtvV303WpcbwnkkLYC0Ft2cYUyHtkstO
fRcRO+K2cZozoSwVPyB8/J9RpcRK3jgnX9lujfwA/pAbP0J2UPQFxmWFRQnFjaq6
rkqbNEBgLy+kFL1NEsRbvFbKrRi5bYy2lNms2NJPZvdNQbT/2dBZKmJqxHkxCuOQ
FjhJQNeO+Njm1Z1iATS/3rts2yZlqXKsxQUzN6vNbD8KnXRMEeOXUYvbV4lqfCf8
mS14WEbSiMy87GB5S9ucSV1XUrlTG5UGcMSZOBcEUpisRPEmQWUOTWIoDQ5FOia/
GI+Ki523r2ruEmbmG37EBSBXdxIdndqrjy+QVAmCebyDx9eVEGOIpn26bW5LKeru
mJxa/CFBaKi4bRvmdJRLAgMBAAGjQjBAMA4GA1UdDwEB/wQEAwIBBjAPBgNVHRMB
Af8EBTADAQH/MB0GA1UdDgQWBBS182Xy/rAKkh/7PH3zRKCsYyXDFDANBgkqhkiG
9w0BAQsFAAOCAgEAncDZNytDbrrVe68UT6py1lfF2h6Tm2p8ro42i87WWyP2LK8Y
nLHC0hvNfWeWmjZQYBQfGC5c7aQRezak+tHLdmrNKHkn5kn+9E9LCjCaEsyIIn2j
qdHlAkepu/C3KnNtVx5tW07e5bvIjJScwkCDbP3akWQixPpRFAsnP+ULx7k0aO1x
qAeaAhQ2rgo1F58hcflgqKTXnpPM02intVfiVVkX5GXpJjK5EoQtLceyGOrkxlM/
sTPq4UrnypmsqSagWV3HcUlYtDinc+nukFk6eR4XkzXBbwKajl0YjztfrCIHOn5Q
CJL6TERVDbM/aAPlv8kJ1sWGLuvvWYzMYqLzDul//rUF10qEMWaXVZV51KpS9DY/
```

The *domain* of the new model registry is contained in the list-model-registries response:



3. Apply ConfigMap Update

1. Update the KUBECONFIG back to the AI Inference service cluster kubeconfig.

2. Edit the ConfigMap of the ML Serving Cluster:

    - **kubectl edit cm modelregistry-config-controlplane -n serving**
        - Update **tls.crt:** with the data from above
    - **kubectl describe cm api-config -n serving**
        - Update **model.registry.url:** with the data from above

3. Restart the deployment to force the cm changes to take effect
    - kubectl scale deployment api -n serving --replicas=0
    - kubectl scale deployment api -n serving --replicas=1