

Machine Learning

Models

Date published: 2020-07-16

Date modified: 2022-04-11

CLOUDERA

<https://docs.cloudera.com/>

Legal Notice

© Cloudera Inc. 2023. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Model Training and Deployment Overview.....	5
Experiments.....	5
Experiments - Concepts and Terminology.....	6
Models.....	7
Models - Concepts and Terminology.....	8
Challenges with Machine Learning in production.....	9
Challenges with model deployment and serving.....	10
Challenges with model monitoring.....	10
Challenges with model governance.....	11
Model visibility.....	12
Model explainability, interpretability, and reproducibility.....	12
Model governance using Apache Atlas.....	12
Creating and Deploying a Model.....	13
Usage Guidelines.....	17
Known Issues and Limitations.....	18
Model Request and Response Formats.....	19
Testing Calls to a Model.....	20
Securing Models.....	22
Access Keys for Models.....	22
API Key for Models.....	22
Enabling authentication.....	23
Generating an API key.....	23
Managing API Keys.....	24
Workflows for Active Models.....	24
Technical Metrics for Models.....	26

Debugging Issues with Models.....	26
Deleting a Model.....	28
Example - Model Training and Deployment (Iris).....	28
Train the Model.....	29
Deploy the Model.....	31

Model Training and Deployment Overview

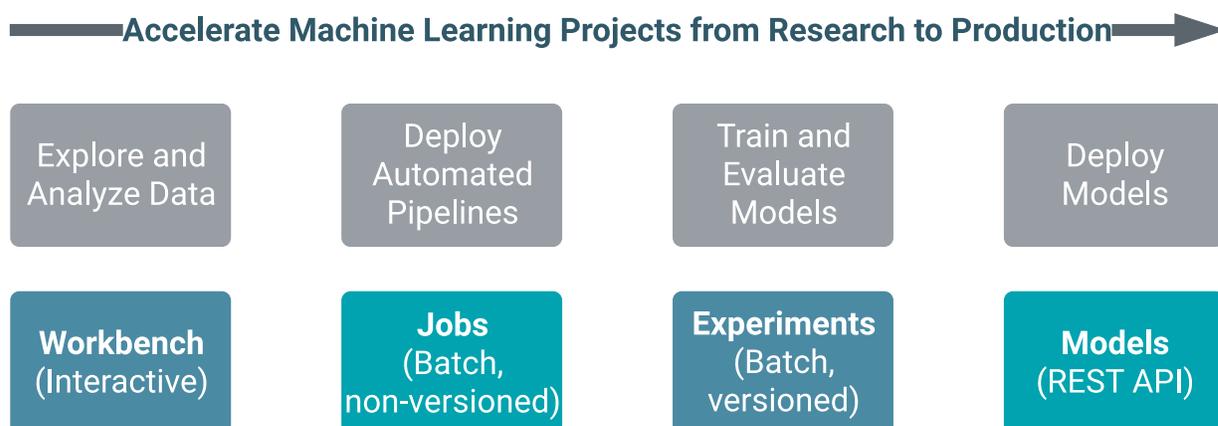
This section provides an overview of model training and deployment using Cloudera Machine Learning.

Machine learning is a discipline that uses computer algorithms to extract useful knowledge from data. There are many different types of machine learning algorithms, and each one works differently. In general however, machine learning algorithms begin with an initial hypothetical model, determine how well this model fits a set of data, and then work on improving the model iteratively. This training process continues until the algorithm can find no additional improvements, or until the user stops the process.

A typical machine learning project will include the following high-level steps that will transform a loose data hypothesis into a model that serves predictions.

1. Explore and experiment with and display findings of data
2. Deploy automated pipelines of analytics workloads
3. Train and evaluate models
4. Deploy models as REST APIs to serve predictions

With Cloudera Machine Learning, you can deploy the complete lifecycle of a machine learning project from research to deployment.



Experiments

This topic introduces you to experiments, and the challenge this feature aims to solve.

Cloudera Machine Learning allows data scientists to run batch experiments that track different versions of code, input parameters, and output (both metrics and files).

Challenge

As data scientists iteratively develop models, they often experiment with datasets, features, libraries, algorithms, and parameters. Even small changes can significantly impact the resulting model. This means data scientists need the ability to iterate and repeat similar experiments in parallel and on demand, as they rely on differences in output and scores to tune parameters until they obtain the best fit for the problem at hand. Such a training workflow requires versioning of the file system, input parameters, and output of each training run.

Without versioned experiments you would need intense process rigor to consistently track training artifacts (data, parameters, code, etc.), and even then it might be impossible to reproduce and explain a given result. This can lead to wasted time and effort during collaboration, not to mention the compliance risks introduced.

Solution

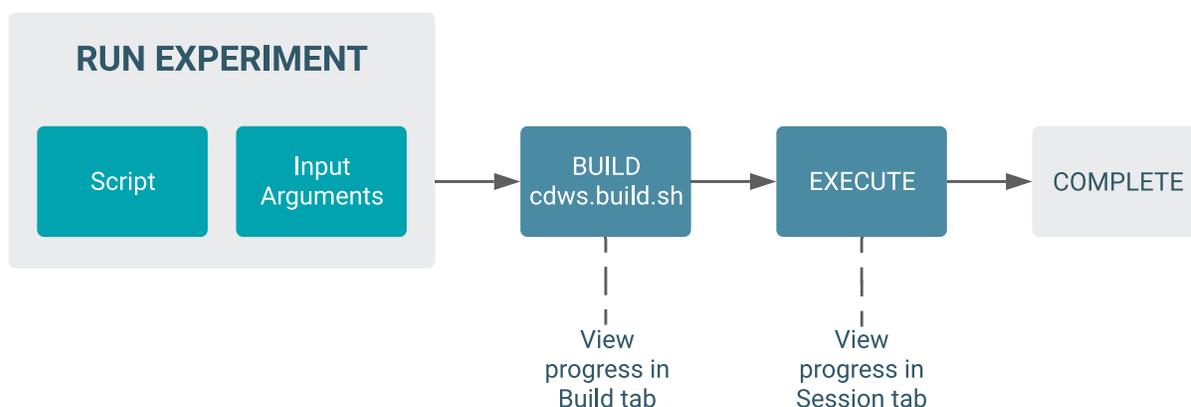
Cloudera Machine Learning uses experiments to facilitate ad-hoc batch execution and model training. Experiments are batch executed workloads where the code, input parameters, and output artifacts are versioned. This feature also provides a lightweight ability to track output data, including files, metrics, and metadata for comparison.

Experiments - Concepts and Terminology

This topic walks you through some basic concepts and terminology related to experiments.

The term experiment refers to a non interactive batch execution script that is versioned across input parameters, project files, and output. Batch experiments are associated with a specific project (much like sessions or jobs) and have no notion of scheduling; they run at creation time. To support versioning of the project files and retain run-level artifacts and metadata, each experiment is executed in an isolated container.

Lifecycle of an Experiment



The rest of this section describes the different stages in the lifecycle of an experiment - from launch to completion.

1. Launch Experiment

In this step you will select a script from your project that will be run as part of the experiment, and the resources (memory/GPU) needed to run the experiment. The engine kernel will be selected by default based on your script. For detailed instructions on how to launch an experiment, see *Getting Started with Cloudera Machine Learning*.

2. Build

When you launch the experiment, Cloudera Machine Learning first builds a new versioned engine image where the experiment will be executed in isolation. This new engine includes:

- the base engine image used by the project (check Project Settings)
- a snapshot of the project filesystem
- environmental variables inherited from the project.
- packages explicitly specified in the project's build script (cdsw-build.sh)

It is your responsibility to provide the complete list of dependencies required for the experiment via the cdsw-build.sh file. As part of the engine's build process, Cloudera Machine Learning will run the cdsw-build.sh script and install the packages or libraries requested there on the new image.

For details about the build process and examples on how to specify dependencies, see [Engines for Experiments and Models](#).

3. Schedule

Once the engine is built the experiment is scheduled for execution like any other job or session. Once the requested CPU/GPU and memory have been allocated to the experiment, it will move on to the execution stage.

Note that if your deployment is running low on memory and CPU, your runs may spend some time in this stage.

4. Execute

This is the stage where the script you have selected will be run in the newly built engine environment. This is the same output you would see if you had executed the script in a session in the Workbench console.

You can watch the execution in progress in the individual run's Session tab.

You can also go to the project Overview Experiments page to see a table of all the experiments launched within that project and their current status.

Run ID: A numeric ID that tracks all experiments launched on a Cloudera Machine Learning deployment. It is not limited to the scope of a single user or project.

Related Information

[Running an Experiment with Cloudera Machine Learning](#)

Models

Cloudera Machine Learning allows data scientists to build, deploy, and manage models as REST APIs to serve predictions.

Challenge

Data scientists often develop models using a variety of Python/R open source packages. The challenge lies in actually exposing those models to stakeholders who can test the model. In most organizations, the model deployment process will require assistance from a separate DevOps team who likely have their own policies about deploying new code.

For example, a model that has been developed in Python by data scientists might be rebuilt in another language by the devops team before it is actually deployed. This process can be slow and error-prone. It can take months to deploy new models, if at all. This also introduces compliance risks when you take into account the fact that the new re-developed model might not be even be an accurate reproduction of the original model.

Once a model has been deployed, you then need to ensure that the devops team has a way to rollback the model to a previous version if needed. This means the data science team also needs a reliable way to retain history of the models they build and ensure that they can rebuild a specific version if needed. At any time, data scientists (or any other stakeholders) must have a way to accurately identify which version of a model is/was deployed.

Solution

Cloudera Machine Learning allows data scientists to build and deploy their own models as REST APIs. Data scientists can now select a Python or R function within a project file, and Cloudera Machine Learning will:

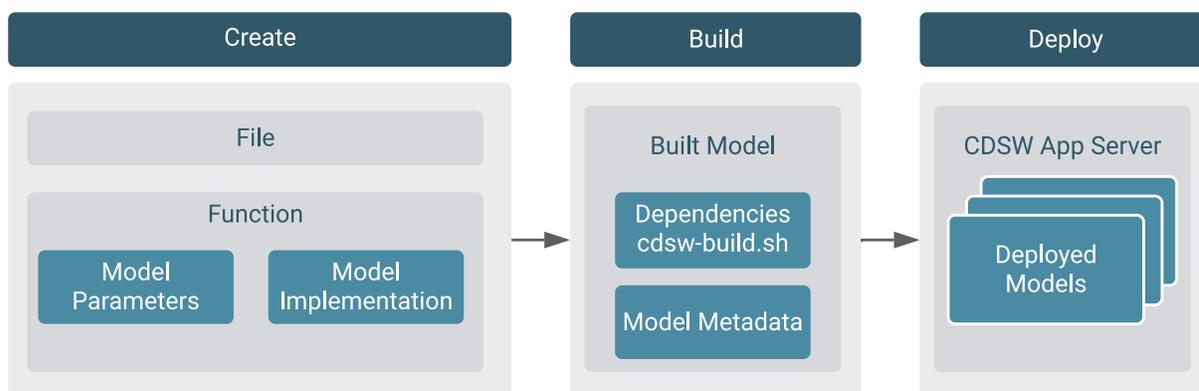
- Create a snapshot of model code, model parameters, and dependencies.
- Package a trained model into an immutable artifact and provide basic serving code.
- Add a REST endpoint that automatically accepts input parameters matching the function, and that returns a data structure that matches the function's return type.
- Save the model along with some metadata.
- Deploy a specified number of model API replicas, automatically load balanced.

Models - Concepts and Terminology

Model

Model is a high level abstract term that is used to describe several possible incarnations of objects created during the model deployment process. For the purpose of this discussion you should note that 'model' does not always refer to a specific artifact. More precise terms (as defined later in this section) should be used whenever possible.

Stages of the Model Deployment Process



The rest of this section contains supplemental information that describes the model deployment process in detail.

Create

- File - The R or Python file containing the function to be invoked when the model is started.
- Function - The function to be invoked inside the file. This function should take a single JSON-encoded object (for example, a python dictionary) as input and return a JSON-encodable object as output to ensure compatibility with any application accessing the model using the API. JSON decoding and encoding for model input/output is built into Cloudera Machine Learning.

The function will likely include the following components:

- Model Implementation

The code for implementing the model (e.g. decision trees, k-means). This might originate with the data scientist or might be provided by the engineering team. This code implements the model's predict function, along with any setup and teardown that may be required.

- Model Parameters

A set of parameters obtained as a result of model training/fitting (using experiments). For example, a specific decision tree or the specific centroids of a k-means clustering, to be used to make a prediction.

Build

This stage takes as input the file that calls the function and returns an artifact that implements a single concrete model, referred to as a model build.

- Built Model

A built model is a static, immutable artifact that includes the model implementation, its parameters, any runtime dependencies, and its metadata. If any of these components need to be changed, for example, code changes to the implementation or its parameters need to be

retrained, a new build must be created for the model. Model builds are versioned using build numbers.

To create the model build, Cloudera Machine Learning creates a Docker image based on the engine designated as the project's default engine. This image provides an isolated environment where the model implementation code will run.

To configure the image environment, you can specify a list of dependencies to be installed in a build script called `cdsw-build.sh`.

For details about the build process and examples on how to install dependencies, see *Engines for Experiments and Models*.

- Build Number:

Build numbers are used to track different versions of builds within the scope of a single model. They start at 1 and are incremented with each new build created for the model.

Deploy

This stage takes as input the memory/CPU resources required to power the model, the number of replicas needed, and deploys the model build created in the previous stage to a REST API.

- Deployed Model

A deployed model is a model build in execution. A built model is deployed in a model serving environment, likely with multiple replicas.

- Environmental Variable

You can set environmental variables each time you deploy a model. Note that models also inherit any environment variables set at the project and global level. (For more information see *Engine Environment Variables*.) However, in case of any conflicts, variables set per-model will take precedence.



Note: If you are using any model-specific environmental variables, these must be specified every time you re-deploy a model. Models do not inherit environmental variables from previous deployments.

- Model Replicas

The engines that serve incoming requests to the model. Note that each replica can only process one request at a time. Multiple replicas are essential for load-balancing, fault tolerance, and serving concurrent requests. Cloudera Machine Learning allows you to deploy a maximum of 9 replicas per model.

- Deployment ID

Deployment IDs are numeric IDs used to track models deployed across Cloudera Machine Learning. They are not bound to a model or project.

Related Information

[Experiments - Concepts and Terminology](#)

[Engines for Experiments and Models](#)

[Engines Environment Variables](#)

Challenges with Machine Learning in production

One of the hardest parts of Machine Learning (ML) is deploying and operating ML models in production applications. These challenges fall mainly into the following categories: model deployment and serving, model monitoring, and model governance.

Challenges with model deployment and serving

After models are trained and ready to deploy in a production environment, lack of consistency with model deployment and serving workflows can present challenges in terms of scaling your model deployments to meet the increasing numbers of ML usecases across your business.

Many model serving and deployment workflows have repeatable, boilerplate aspects which you can automate using modern DevOps techniques like high frequency deployment and microservices architectures. This approach can enable the ML engineers to focus on the model instead of the surrounding code and infrastructure.

Challenges with model monitoring

Machine Learning (ML) models predict the world around them which is constantly changing. The unique and complex nature of model behavior and model lifecycle present challenges after the models are deployed.

Cloudera Machine Learning provides you the capability to monitor the performance of the model on two levels: technical performance (latency, throughput, and so on similar to an [Application Performance Management](#)), and mathematical performance (is the model predicting correctly, is the model biased, and so on).

There are two types of metrics that are collected from the models:

- Time series metrics: Metrics measured in-line with model prediction. It can be useful to track the changes in these values over time. It is the finest granular data for the most recent measurement. To improve performance, older data is aggregated to reduce data records and storage.
- Post-prediction metrics: Metrics that are calculated after prediction time, based on ground truth and/or batches (aggregates) of time series metrics. To collect metrics from the models, the Python SDK has been extended to include the following functions that you can use to store different types of metrics:

To collect metrics from the models, the Python SDK has been extended to include the following functions that you can use to store different types of metrics:

- `track_metrics`: Tracks the metrics generated by experiments and models.
- `read_metrics`: Reads the metrics already tracked for a deployed model, within a given window of time.
- `track_delayed_metrics`: Tracks metrics that correspond to individual predictions, but aren't known at the time the prediction is made. The most common instances are ground truth and metrics derived from ground truth such as error metrics.
- `track_aggregate_metrics`: Registers metrics that are not associated with any particular prediction. This function can be used to track metrics accumulated and/or calculated over a longer period of time.

The following two use-cases show how you can use these functions:

- Tracking accuracy of a model over time
- Tracking drift

Usecase 1: Tracking accuracy of a model over time

Consider the case of a large telco. When a customer service representative takes a call from a customer, a web application presents an estimate of the risk that the customer will churn. The service representative takes this risk into account when evaluating whether to offer promotions.

The web application obtains the risk of churn by calling into a model hosted on Cloudera Machine Learning (CML). For each prediction thus obtained, the web application records the UUID into a datastore alongside the customer ID. The prediction itself is tracked in CML using the `track_metrics` function.

At some point in the future, some customers do in fact churn. When a customer churns, they or another customer service representative close their account in a web application. That web application records the churn event, which is ground truth for this example, in a datastore.

An ML engineer who works at the telco wants to continuously evaluate the suitability of the risk model. To do this, they create a recurring CML job. At each run, the job uses the `read_metrics` function to read all the predictions that

were tracked in the last interval. It also reads in recent churn events from the ground truth datastore. It joins the churn events to the predictions and customer ID's using the recorded UUID's, and computes an Receiver operating characteristic (ROC) metric for the risk model. The ROC is tracked in the metrics store using the `track_aggregate_metrics` function.



Note: You can store the ground truth in an external datastore, such as Cloudera Data Warehouse or in the metrics store.

Use-case 2: Tracking drift

Instead of or in addition to computing ROC, the ML engineer may need to track various types of drift. Drift metrics are especially useful in cases where ground truth is unavailable or is difficult to obtain.

The definition of drift is broad and somewhat nebulous and practical approaches to handling it are evolving, but drift is always about changing distributions. The distribution of the input data seen by the model may change over time and deviate from the distribution in the training dataset, and/or the distribution of the output variable may change, and/or the relationship between input and output may change.

All drift metrics are computed by aggregating batches of predictions in some way. As in the use case above, batches of predictions can be read into recurring jobs using the `read_metrics` function, and the drift metrics computed by the job can be tracked using the `track_aggregate_metrics` function.

Challenges with model governance

Businesses implement ML models across their entire organization, spanning a large spectrum of usecases. When you start deploying more than just a couple models in production, a lot of complex governance and management challenges arise.

Almost all the governance needs for ML are associated with data and are tied directly to the data management practice in your organization. For example, what data can be used for certain applications, who should be able to access what data, and based on what data are models created.

Some of the other unique governance challenges that you could encounter are:

- How to gain visibility into the impact your models have on your customers?
- How can you ensure you are still compliant with both governmental and internal regulations?

- How does your organization's security practices apply to the models in production?

Ultimately, the needs for ML governance can be distilled into the following key areas: model visibility, and model explainability, interpretability, and reproducibility.

Model visibility

A basic requirement for model governance is enabling teams to understand how machine learning is being applied in their organizations. This requires a canonical catalog of models in use. In the absence of such a catalog, many organizations are unaware of how their models work, where they are deployed, what they are being used for, and so on. This leads to repeated work, model inconsistencies, recomputing features, and other inefficiencies.

Model explainability, interpretability, and reproducibility

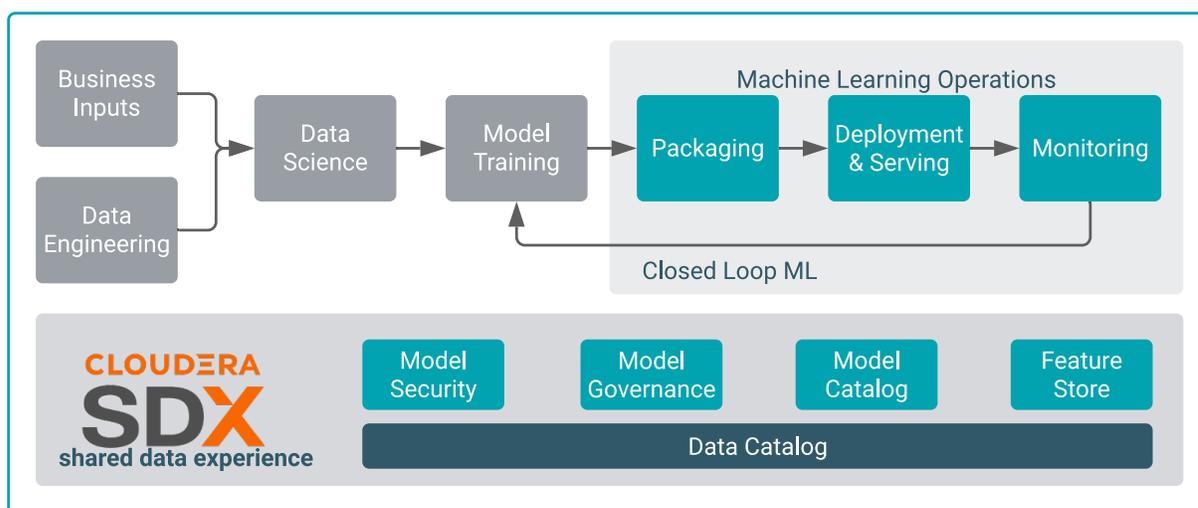
Models are often seen as a black box: data goes in, something happens, and a prediction comes out. This lack of transparency is challenging on a number of levels and is often represented in loosely related terms explainability, interpretability, and reproducibility.

- Explainability: Indicates the description of the internal mechanics of an Machine Learning (ML) model in human terms
- Interpretability: Indicates the ability to:
 - Understand the relationship between model inputs, features and outputs
 - Predict the response to changes in inputs
- Reproducibility: Indicates the ability to reproduce the output of a model in a consistent fashion for the same inputs

To solve these challenges, CML provides an end-to-end model governance and monitoring workflow that gives organizations increased visibility into their machine learning workflows and aims to eliminate the blackbox nature of most machine learning models.

The following image shows the end-to-end production ML workflow:

Figure 1: Production ML Workflow



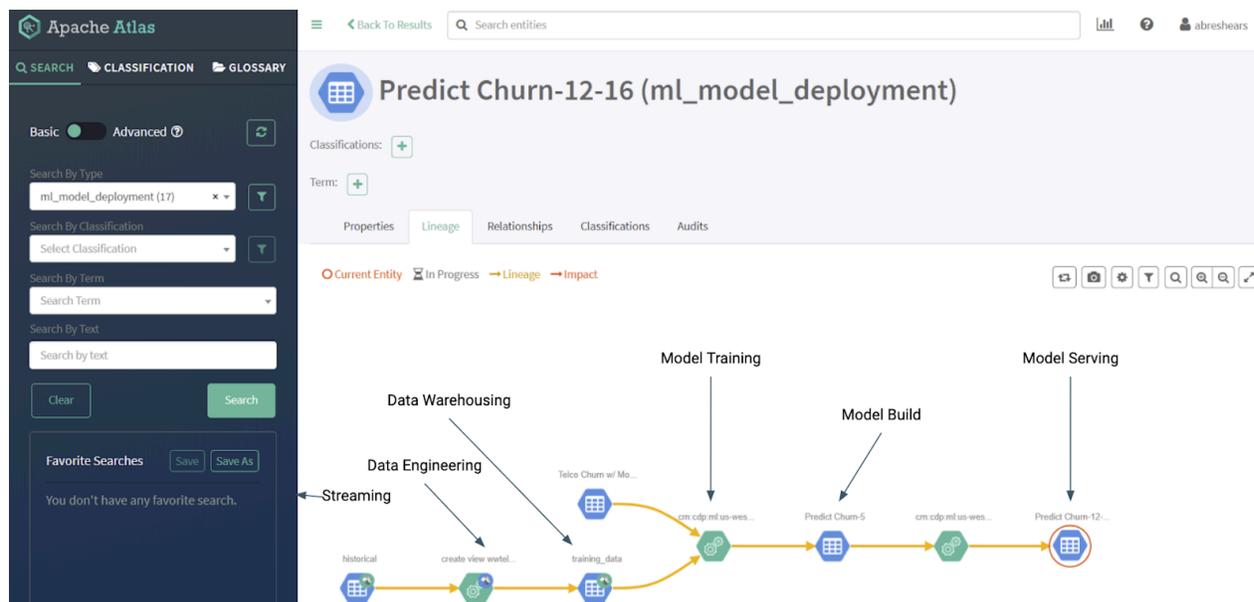
Model governance using Apache Atlas

To address governance challenges, Cloudera Machine Learning uses Apache Atlas to automatically collect and visualize lineage information for data used in Machine Learning (ML) workflows — from training data to model deployments.

Lineage is a visual representation of the project. The lineage information includes visualization of the relationships between model entities such as code, model builds, deployments, and so on, and the processes that carry out transformations on the data involved, such as create project, build model, deploy model, and so on.

The Apache Atlas type system has pre-built governance features that can be used to define ML metadata objects. A type in Atlas is a definition of the metadata stored to describe a data asset or other object or process in an environment. For ML governance, Cloudera has designed new Atlas types that capture ML entities and processes as Atlas metadata objects.

In addition to the definition of the types, Atlas also captures the relationship between the entities and processes to visualize the end-to-end lineage flow, as shown in the following image. The blue hexagons represent an entity (also called the noun) and the green hexagons represent a process (also called the verb).



The ML metadata definition closely follows the actual machine learning workflow. Training data sets are the starting point for a model lineage flow. These data sets can be tables from a data warehouse or an embedded csv file. Once a data set has been identified, the lineage follows into training, building and deploying the model.

See *ML operations entities created in Atlas* for a list of metadata that Atlas collects from each CML workspace. Metadata is collected from machine learning projects, model builds, and model deployments, and the processes that create these entities.

Creating and Deploying a Model

This topic describes a simple example of how to create and deploy a model using Cloudera Machine Learning.

Using Cloudera Machine Learning, you can create any function within a script and deploy it to a REST API. In a machine learning project, this will typically be a predict function that will accept an input and return a prediction based on the model's parameters.

For the purpose of this quick start demo we are going to create a very simple function that adds two numbers and deploy it as a model that returns the sum of the numbers. This function will accept two numbers in JSON format as input and return the sum.

For CML UI

1. Create a new project. Note that models are always created within the context of a project.
2. Click New Session and launch a new Python 3 session.

3. Create a new file within the project called `add_numbers.py`. This is the file where we define the function that will be called when the model is run. For example:

`add_numbers.py`

```
def add(args):  
    result = args["a"] + args["b"]  
    return result
```



Note: In practice, do not assume that users calling the model will provide input in the correct format or enter good values. Always perform input validation.

4. Before deploying the model, test it by running the `add_numbers.py` script, and then calling the `add` function directly from the interactive workbench session. For example:

```
add({ "a": 3, "b": 5 })
```

The screenshot displays a JupyterLab environment. On the left, a code editor shows the following Python code:

```
1 # Function to add two numbers  
2  
3 def add(args):  
4     result = args["a"] + args["b"]  
5     return result  
6
```

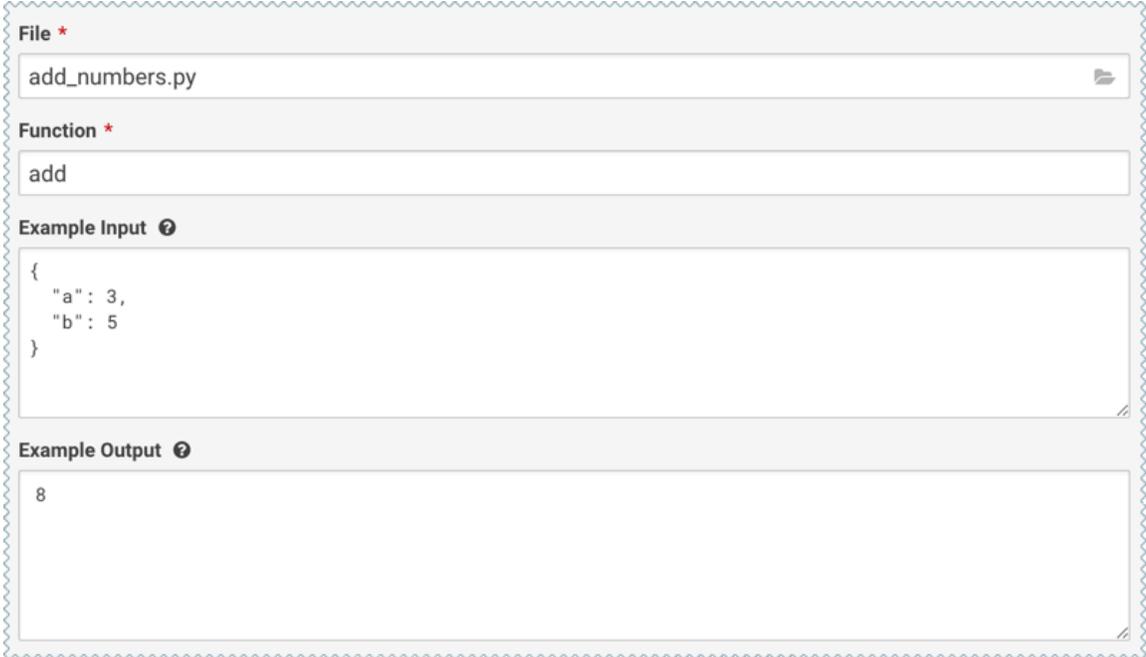
On the right, a terminal window titled "Untitled Session" is shown. It displays the execution of the `add` function with two different inputs:

```
> add({"a": 3, "b": 5})  
8  
> add({"a": 4, "b": 7})  
11
```

The terminal window also shows session details: "By [user] - Python 3 Session - 1 vCPU / 2 GiB Memory - 50 minutes ago" and a "Running" status.

5. Deploy the add function to a REST endpoint.

- a. Go to the project Overview page.
- b. Click Models New Model .
- c. Give the model a Name and Description.
- d. Enter details about the model that you want to build. In this case:
 - File: add_numbers.py
 - Function: add
 - Example Input: {"a": 3, "b": 5}
 - Example Output: 8



The screenshot shows a configuration form for a new model. It has four main sections:

- File ***: A text input field containing "add_numbers.py".
- Function ***: A text input field containing "add".
- Example Input ?**: A text area containing a JSON object:

```
{  
  "a": 3,  
  "b": 5  
}
```
- Example Output ?**: A text area containing the number "8".

- e. Select the resources needed to run this model, including any replicas for load balancing.



Note: The list of options here is specific to the default engine you have specified in your Project Settings: ML Runtimes or Legacy Engines. Engines allow kernel selection, while ML Runtimes allow Editor, Kernel, Variant, and Version selection. Resource Profile list is applicable for both ML Runtimes and Legacy Engines.

- f. Click Deploy Model.

- Click on the model to go to its Overview page. Click Builds to track realtime progress as the model is built and deployed. This process essentially creates a Docker container where the model will live and serve requests.

- Once the model has been deployed, go back to the model Overview page and use the Test Model widget to make sure the model works as expected.

If you entered example input when creating the model, the Input field will be pre-populated with those values. Click Test. The result returned includes the output response from the model, as well as the ID of the replica that served the request.

Model response times depend largely on your model code. That is, how long it takes the model function to perform the computation needed to return a prediction. It is worth noting that model replicas can only process one request at a time. Concurrent requests will be queued until the model can process them.

For CML APIv2

To create and deploy a model using the API, follow this example:

This example demonstrates the use of the Models API. To run this example, first do the following:

- Create a project with the Python template and a legacy engine.
- Start a session.
- Run `!pip3 install sklearn`
- Run `fit.py`

The example script first obtains the project ID, then creates and deploys a model.

```
projects = client.list_projects(search_filter=json.dumps({"name": "<your
project name>}))
project = projects.projects[0] # assuming only one project is returned by
the above query
model_body = cmlapi.CreateModelRequest(project_id=project.id, name="Demo
Model", description="A simple model")
model = client.create_model(model_body, project.id)
model_build_body = cmlapi.CreateModelBuildRequest(project_id=project.id,
model_id=model.id, file_path="predict.py", function_name="predict", ker
nel="python3")
model_build = client.create_model_build(model_build_body, project.id, mod
el.id)
while model_build.status not in ["built", "build failed"]:
    print("waiting for model to build...")
    time.sleep(10)
    model_build = client.get_model_build(project.id, model.id, model_build
.id)
if model_build.status == "build failed":
```

```
print("model build failed, see UI for more information")
sys.exit(1)
print("model built successfully!")
model_deployment_body = cmlapi.CreateModelDeploymentRequest(project_id=p
roject.id, model_id=model.id, build_id=model_build.id)
model_deployment = client.create_model_deployment(model_deployment_body,
project.id, model.id, build.id)
while model_deployment.status not in ["stopped", "failed", "deployed"]:
    print("waiting for model to deploy...")
    time.sleep(10)
model_deployment = client.get_model_deployment(project.id, model.id, m
odel_build.id, model_deployment.id)
if model_deployment.status != "deployed":
    print("model deployment failed, see UI for more information")
    sys.exit(1)
print("model deployed successfully!")
```

Usage Guidelines

This section calls out some important guidelines you should keep in mind when you start deploying models with Cloudera Machine Learning.

Model Code

Models in Cloudera Machine Learning are designed to run any code that is wrapped into a function. This means you can potentially deploy a model that returns the result of a `SELECT *` query on a very large table. However, Cloudera strongly recommends against using the models feature for such use cases.

As a best practice, your models should be returning simple JSON responses in near-real time speeds (within a fraction of a second). If you have a long-running operation that requires extensive computing and takes more than 15 seconds to complete, consider using batch jobs instead.

Model Artifacts

Once you start building larger models, make sure you are storing these model artifacts in HDFS, S3, or any other external storage. Do not use the project filesystem to store large output artifacts.

In general, any project files larger than 50 MB must be part of your project's `.gitignore` file so that they are not included in *Engines for Experiments and Models* for future experiments/model builds. Note that in case your models require resources that are stored outside the model itself, it is up to you to ensure that these resources are available and immutable as model replicas may be restarted at any time.

Resource Consumption and Scaling

Models should be treated as any other long-running applications that are continuously consuming memory and computing resources. If you are unsure about your resource requirements when you first deploy the model, start with a single replica, monitor its usage, and scale as needed.

If you notice that your models are getting stuck in various stages of the deployment process, check the *Monitoring Active Models* page to make sure that the cluster has sufficient resources to complete the deployment operation.

Security Considerations

As stated previously, models do not impose any limitations on the code they can run. Additionally, models run with the permissions of the user that creates the model (same as sessions and jobs).

Therefore, be conscious of potential data leaks especially when querying underlying data sets to serve predictions.

Cloudera Machine Learning models are not public by default. Each model has an access key associated with it. Only users/applications who have this key can make calls to the model. Be careful with who has permission to view this key.

Cloudera Machine Learning also prints stderr/stdout logs from models to an output pane in the UI. Make sure you are not writing any sensitive information to these logs.

Deployment Considerations

Models deployed using Cloudera Machine Learning in the public cloud are highly available subject to the following limitations:

- Model high availability is dependent on the high availability of the cloud provider's Kubernetes service. Please refer to your chosen cloud provider for precise SLAs.
- Model high availability is dependent on the high availability of the cloud provider's load balancer service. Please refer to your chosen cloud provider for precise SLAs.
- In the event that the Kubernetes pod running the model proxy service becomes unavailable, the Model may be unavailable for multiple seconds during failover.

There can only be one active deployment per model at any given time. This means you should plan for model downtime if you want to deploy a new build of the model or re-deploy with more or fewer replicas.

Keep in mind that models that have been developed and trained using Cloudera Machine Learning are essentially Python or R code that can easily be persisted and exported to external environments using popular serialization formats such as Pickle, PMML, ONNX, and so on.

Related Information

[Engines for Experiments and Models](#)

[Technical Metrics for Models](#)

Known Issues and Limitations

- Known Issues with Model Builds and Deployed Models
 - Re-deploying or re-building models results in model downtime (usually brief).
 - Re-starting Cloudera Machine Learning does not automatically restart active models. These models must be manually restarted so they can serve requests again.

Cloudera Bug: DSE-4950

- Model deployment will fail if your project filesystem is too large for the Git *Engines for Experiments and Models* process. As a general rule, any project files (code, generated model artifacts, dependencies, etc.) larger

than 50 MB must be part of your project's .gitignore file so that they are not included in snapshots for model builds.

- Model builds will fail if your project filesystem includes a .git directory (likely hidden or nested). Typical build stage errors include:

```
Error: 2 UNKNOWN: Unable to schedule build: [Unable to create a checkpoint of current source: [Unable to push sources to git server: ...
```

To work around this, rename the .git directory (for example, NO.git) and re-build the model.

Cloudera Bug: DSE-4657

- JSON requests made to active models should not be more than 5 MB in size. This is because JSON is not suitable for very large requests and has high overhead for binary objects such as images or video. Call the model with a reference to the image or video, such as a URL, instead of the object itself.
- Any external connections, for example, a database connection or a Spark context, must be managed by the model's code. Models that require such connections are responsible for their own setup, teardown, and refresh.
- Model logs and statistics are only preserved so long as the individual replica is active. Cloudera Machine Learning may restart a replica at any time it is deemed necessary (such as bad input to the model).
- (MLLib) The MLLib model.save() function fails with the following sample error. This occurs because the Spark executors on CML all share a mount of /home/cdsw which results in a race condition as multiple executors attempt to write to it at the same time.

```
Caused by:
           java.io.IOException: Mkdirs failed to create
           file:/home/cdsw/model.mllib/metadata/_temporary ....
```

Recommended workarounds:

- Save the model to /tmp, then move it into /home/cdsw on the driver/session.
- Save the model to either an S3 URL or any other explicit external URL.
- Limitations
 - Scala models are not supported.
 - Spawning worker threads are not supported with models.
 - Models deployed using Cloudera Machine Learning in the public cloud are highly available subject to the following limitations:
 - Model high availability is dependent on the high availability of the cloud provider's Kubernetes service. Please refer to your chosen cloud provider for precise SLAs.
 - Model high availability is dependent on the high availability of the cloud provider's load balancer service. Please refer to your chosen cloud provider for precise SLAs.
 - In the event that the Kubernetes pod running the model proxy service becomes unavailable, the Model may be unavailable for multiple seconds during failover.
 - Dynamic scaling and auto-scaling are not currently supported. To change the number of replicas in service, you will have to re-deploy the build.

Related Information

[Engines for Experiments and Models](#)

[Distributed Computing with Workers](#)

Model Request and Response Formats

Every model function in Cloudera Machine Learning takes a single argument in the form of a JSON-encoded object, and returns another JSON-encoded object as output. This format ensures compatibility with any application accessing the model using the API, and gives you the flexibility to define how JSON data types map to your model's datatypes.

Model Requests

When making calls to a model, keep in mind that JSON is not suitable for very large requests and has high overhead for binary objects such as images or video. Consider calling the model with a reference to the image or video such as a URL instead of the object itself. Requests to models should not be more than 5 MB in size. Performance may degrade and memory usage increase for larger requests.

Ensure that the JSON request represents all objects in the request or response of a model call. For example, JSON does not natively support dates. In such cases consider passing dates as strings, for example in ISO-8601 format, instead.

For a simple example of how to pass JSON arguments to the model function and make calls to deployed model, see *Creating and Deploying a Model*.

Model Responses

Models return responses in the form of a JSON-encoded object. Model response times depend on how long it takes the model function to perform the computation needed to return a prediction. Model replicas can only process one request at a time. Concurrent requests are queued until a replica is available to process them.

When Cloudera Machine Learning receives a call request for a model, it attempts to find a free replica that can answer the call. If the first arbitrarily selected replica is busy, Cloudera Machine Learning will keep trying to contact a free replica for 30 seconds. If no replica is available, Cloudera Machine Learning will return a `model.busy` error with HTTP status code 429 (Too Many Requests). If you see such errors, re-deploy the model build with a higher number of replicas.

Model request timeout

You can set the model request timeout duration to a custom value. The default value is 30 seconds. The timeout can be changed if model requests might take more than 30 seconds.

To set the timeout value:

1. As an Admin user, open a CLI.
2. At the prompt, execute the following command. Substitute `<value>` with the number of seconds to set.

```
kubectl set env deployment model-proxy MODEL_REQUEST_TIMEOUT_SECONDS=<value> -n mlx
```

This edits the `kubeconfig` file and sets a new value for the timeout duration.

Related Information

[Creating and Deploying a Model](#)

[Workflows for Active Models](#)

Testing Calls to a Model

Cloudera Machine Learning provides two ways to test calls to a model:

- Test Model Widget

On each model's Overview page, Cloudera Machine Learning provides a widget that makes a sample call to the deployed model to ensure it is receiving input and returning results as expected.

Test Model

Input

```
{
  "a": 3,
  "b": 5
}
```

Test Reset

Result

Status	● success
Response	8
Replica ID	add-two-numbers-1-1-86b9b58b7b-g6s8r

- Sample Request Strings

On the model Overview page, Cloudera Machine Learning also provides sample curl and POST request strings that you can use to test calls to the model. Copy/paste the curl request directly into a Terminal to test the call.

Note that these sample requests already include the example input values you entered while building the model, and the access key required to query the model.

Add Two Numbers

[Overview](#)
[Deployments](#)
[Builds](#)
[Monitoring](#)
[Settings](#)

Description

Sample Code

Add two numbers.

Shell
Python
R

```
curl -H "Content-Type: application/json" -H "Authorization: Bearer
<place API key here>" -X POST https://
/endpoint/model -d '{"accessKey":"mgc4w3rdi4
3x28fy4h8e8:", "request": {"a":3, "b":5}}'
```

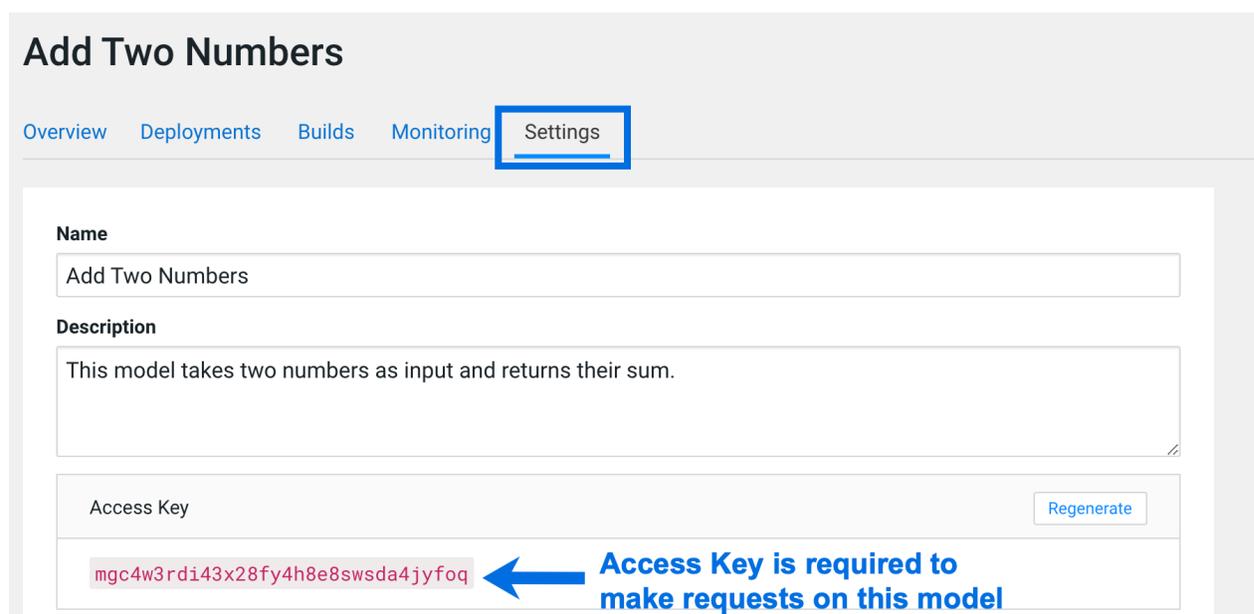
Securing Models

Access Keys for Models

Each model in Cloudera Machine Learning has a unique access key associated with it. This access key is a unique identifier for the model.

Models deployed using Cloudera Machine Learning are not public. In order to call an active model your request must include the model's access key for authentication (as demonstrated in the sample calls above).

To locate the access key for a model, go to the model Overview page and click Settings.



The screenshot shows the 'Add Two Numbers' model settings page. The 'Settings' tab is selected and highlighted with a blue box. Below the tabs, there are three sections: 'Name' (Add Two Numbers), 'Description' (This model takes two numbers as input and returns their sum.), and 'Access Key' (mgc4w3rdi43x28fy4h8e8swwda4jyfoq). A blue arrow points from the text 'Access Key is required to make requests on this model' to the access key field. A 'Regenerate' button is visible next to the access key field.



Important:

Only one access key per model is active at any time. If you regenerate the access key, you will need to re-distribute this access key to users/applications using the model.

Alternatively, you can use this mechanism to revoke access to a model by regenerating the access key. Anyone with an older version of the key will not be able to make calls to the model.

API Key for Models

You can prevent unauthorized access to your models by specifying an API key in the “Authorization” header of your model HTTP request. This topic covers how to create, test, and use an API key in Cloudera Machine Learning.

The API key governs the authentication part of the process and the authorization is based on what privileges the users already have in terms of the project that they are a part of. For example, if a user or application has read-only access to a project, then the authorization is based on their current access level to the project, which is “read-only”. If the users have been authenticated to a project, then they can make a request to a model with the API key. This is different from the previously described Access Key, which is only used to identify which model should serve a request.

Enabling authentication

Restricting access using API keys is an optional feature. By default, the “Enable Authentication” option is turned on. However, it is turned off by default for the existing models for backward compatibility. You can enable authentication for all your existing models.

To enable authentication, go to **Projects Models Settings** and check the **Enable Authentication** option.



Note: It can take up to five minutes for the system to update.

Generating an API key

If you have enabled authentication, then you need an API key to call a model. If you are not a collaborator on a particular project, then you cannot access the models within that project using the API key that you generate. You need to be added as a collaborator by the admin or the owner of the project to use the API key to access a model.

About this task

There are two types of API keys used in Cloudera Machine Learning:

- **API Key:** These are used to authenticate requests to a model. You can choose the expiration period and delete them when no longer needed.
- **Legacy API Key:** This is used in the CDSW-specific internal APIs for CLI automation. This can't be deleted and neither does it expire. This API Key is not required when sending requests to a model.

You can generate more than one API keys to use with your model, depending on the number of clients that you are using to call the models.

Procedure

1. Sign in to Cloudera Machine Learning.
2. Click **Settings** from the left navigation pane.
3. On the **User Settings** page, click the **API Keys** tab.
4. Select an expiry date for the **Model API Key**, and click **Create API keys**.

An API key is generated along with a **Key ID**.

If you do not specify an expiry date, then the generated key is active for one year from the current date, or for the duration set by the Administrator. If you specify an expiration date that exceeds the duration value set by the Administrator, you will get an error. The Administrator can set the default duration value at **Admin Security Default API keys expiration in days**



Note:

- The API key is private and ephemeral. Copy the key and the corresponding key ID on to a secure location for future use before refreshing or leaving the page. If you miss storing the key, then you can generate another key.
- You can delete the API keys that have expired or no longer in use. It can take up to five minutes by the system to take effect.

5. To test the API key:
 - a) Navigate to your project and click Models from the left navigation pane.
 - b) On the **Overview** page, paste the API key in the API key field that you had generated in the previous step and click Test.

The test results, along with the HTTP response code and the Replica ID are displayed in the Results table.

If the test fails and you see the following message, then you must get added as a collaborator on the respective project by the admin or the creator of the project:

```
"User APikey not authorized to access model": "Check APIKEY permissions or model authentication permissions"
```

Managing API Keys

The admin user can access the list of all the users who are accessing the workspace and can delete the API keys for a user.

About this task

To manage users and their keys:

Procedure

1. Sign in to Cloudera Machine Learning as an admin user.
2. From the left navigation pane, click Admin.
The **Site Administration** page is displayed.
3. On the **Site Administration** page, click on the Users tab.
All the users signed under this workspace are displayed.
The API Keys column displays the number of API keys granted to a user.
4. To delete a API key for a particular user:
 - a) Select the user for which you want to delete the API key.
A page containing the user's information is displayed.
 - b) To delete a key, click Delete under the Action column corresponding to the Key ID.
 - c) Click Delete all keys to delete all the keys for that user.



Note: It can take up to five minutes by the system to take effect.

As a non-admin user, you can delete your own API key by navigating to [Settings User Settings API Keys](#) .

Workflows for Active Models

This topic walks you through some nuances between the different workflows available for re-deploying and re-building models.

Active Model - A model that is in the Deploying, Deployed, or Stopping stages.

You can make changes to a model even after it has been deployed and is actively serving requests. Depending on business factors and changing resource requirements, such changes will likely range from changes to the model code itself, to simply modifying the number of CPU/GPUs requested for the model. In addition, you can also stop and restart active models.

Depending on your requirement, you can perform one of the following actions:

Re-deploy an Existing Build

Re-deploying a model involves re-publishing a previously-deployed model in a new serving environment - this is, with an updated number of replicas or memory/CPU/GPU allocation. For example, circumstances that require a re-deployment might include:

- An active model that previously requested a large number of CPUs/GPUs that are not being used efficiently.
- An active model that is dropping requests because it is falling short of replicas.
- An active model needs to be rolled back to one of its previous versions.



Warning: Currently, Cloudera Machine Learning only allows one active deployment per model. This means when you re-deploy a build, the current active deployment will go offline until the re-deployment process is complete and the new deployment is ready to receive requests. Prepare for model downtime accordingly.

To re-deploy an existing model:

1. Go to the model Overview page.
2. Click Deployments.
3. Select the version you want to deploy and click Re-deploy this Build.



Note:

The screenshot shows the Cloudera Machine Learning interface for a model named "Add Two Numbers". At the top right, there are buttons for "Deployed", "Stop", "Restart", and "Deploy New Build". Below these are tabs for "Overview", "Deployments", "Builds", "Monitoring", and "Settings". The "Deployments" tab is active, showing a table with the following data:

Id	Build	Status	Deployed At	Stopped At	Deployed By	Model
4	3	Deployed	Oct 20, 2021, 03:34 PM		csso_wclemens	Id: 2 Name: Add Two Numbers Description: Add two numbers.
3	2	Stopped	Oct 20, 2021, 03:16 PM	Oct 20, 2021, 03:33 PM	csso_wclemens	

A blue box highlights the "Re-deploy This Build" button in the right-hand column of the table.

4. Modify the model serving environment as needed.
5. Click Deploy Model.

Deploy a New Build for a Model

Deploying a new build for a model involves both, re-building the Docker image for the model, and deploying this new build. Note that this is not required if you only need to update the resources allocated to the model. As an example, changes that require a new build might include:

- Code changes to the model implementation.
- Renaming the function that is used to invoke the model.

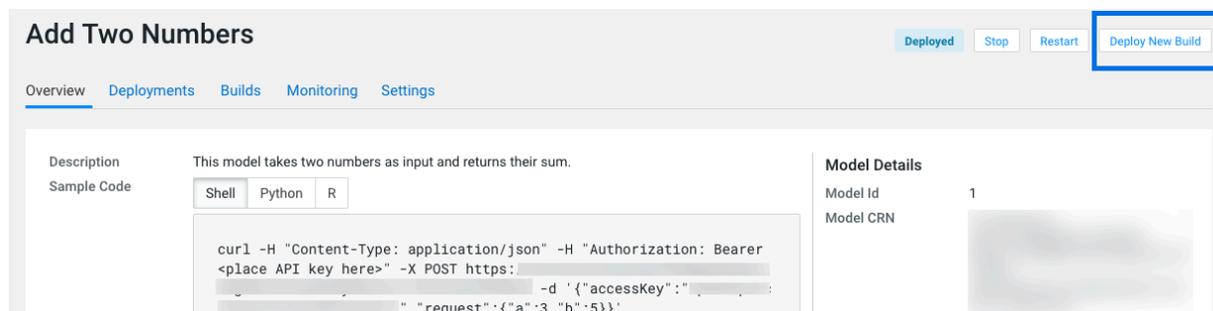


Warning: Currently, Cloudera Machine Learning does not allow you to create a new build for a model without also deploying it. This combined with the fact that you can only have one active deployment per model means that once the new model is built, the current active deployment will go offline so that the new build can be deployed. Prepare for model downtime accordingly.

To create a new build and deploy it:

1. Go to the model Overview page.

2. Click Deploy New Build.



3. Complete the form and click Deploy Model.

Stop a Model

To stop a model (all replicas), go to the model Overview page and click Stop. Click OK to confirm.

Restart a Model

To restart a model (all replicas), go to the model Overview page and click Restart. Click OK to confirm.

Restarting a model does not let you make any code changes to the model. It should primarily be used as a way to quickly re-initialize or re-connect to resources.

Technical Metrics for Models

You can observe the operation of your models by using charts provided for technical metrics. These charts can help you determine if your models are under- or over-resourced, or are experiencing some problem.

To check the performance of your model, go to Models, click on the model name, and select the Monitoring tab. You can choose to monitor all replicas of the model, or choose a specific replica. You can also select the time and date range to display. Up to two weeks of data is retained.

This tab displays charts for the following technical metrics:

- Requests per Second
- Number of Requests
- Number of Failed Requests
- Model Response Time
- All Model Replica CPU Usage
- All Model Replica Memory Usage
- Model Request & Response Size

All charts share a common time axis (the x axis), so it is easy to correlate cpu and memory usage with model response time or the number of failed requests, for example.

Debugging Issues with Models

This topic describes some common issues to watch out for during different stages of the model build and deployment process.

As a general rule, if your model spends too long in any of the afore-mentioned stages, check the resource consumption statistics for the cluster. When the cluster starts to run out of resources, often models will spend some time in a queue before they can be executed.

Resource consumption by active models on a deployment can be tracked by site administrators on the Admin Models page.

Building

Live progress for this stage can be tracked on the model's Build tab. It shows the details of the build process that creates a new Docker image for the model. Potential issues:

- If you specified a custom build script (cdsw-build.sh), ensure that the commands inside the script complete successfully.
- If you are in an environment with restricted network connectivity, you might need to manually upload dependencies to your project and install them from local files.

Pushing

Once the model has been built, it is copied to an internal Docker registry to make it available to all the Cloudera Machine Learning hosts. Depending on network speeds, your model may spend some time in this stage.

Deploying

If you see issues occurring when Cloudera Machine Learning is attempting to start the model, use the following guidelines to begin troubleshooting:

- Make sure your model code works in a workbench session. To do this, launch a new session, run your model file, and then interactively call your target function with the input object. For a simple example, see the *Creating and Deploying a Model*.
- Ensure that you do not have any syntax errors. For Python, make sure you have the kernel with the appropriate Python version (Python 2 or Python 3) selected for the syntax you have used.
- Make sure that your cdsw-build.sh file provides a complete set of dependencies. Dependencies manually installed during a session on the workbench are not carried over to your model. This is to ensure a clean, isolated, build for each model.
- If your model accesses resources such as data on the CDH cluster or an external database make sure that those resources can accept the load your model may exert on them.

Deployed

Once a model is up and running, you can track some basic logs and statistics on the model's Monitoring page. In case issues arise:

- Check that you are handling bad input from users. If your function throws an exception, Cloudera Machine Learning will restart your model to attempt to get back to a known good state. The user will see an unexpected model shutdown error.

For most transient issues, model replicas will respond by restarting on their own before they actually crash. This auto-restart behavior should help keep the model online as you attempt to debug runtime issues.

- Make runtime troubleshooting easier by printing errors and output to stderr and stdout. You can catch these on each model's Monitoring tab. Be careful not to log sensitive data here.
- The Monitoring tab also displays the status of each replica and will show if the replica cannot be scheduled due to a lack of cluster resources. It will also display how many requests have been served/dropped by each replica.

Related Information

[Engines for Experiments and Models](#)

[Creating and Deploying a Model](#)

[Technical Metrics for Models](#)

Deleting a Model

Before you begin



Important:

- You must stop all active deployments before you delete a model. If not stopped, active models will continue serving requests and consuming resources even though they do not show up in Cloudera Machine Learning UI.
- Deleted models are not actually removed from disk. That is, this operation will not free up storage space.

Procedure

1. Go to the model [Overview Settings](#) .
2. Click [Delete Model](#).

Deleting a model removes all of the model's builds and its deployment history from Cloudera Machine Learning.

You can also delete specific builds from a model's history by going to the model's [Overview Build](#) page.

Example - Model Training and Deployment (Iris)

This topic uses Cloudera Machine Learning's built-in Python template project to walk you through an end-to-end example where we use experiments to develop and train a model, and then deploy it using Cloudera Machine Learning.

This example uses the canonical [Iris](#) dataset from [Fisher and Anderson](#) to build a model that predicts the width of a flower's petal based on the petal's length.

The scripts for this example are available in the Python template project that ships with Cloudera Machine Learning. First, create a new project from the Python template:

Create a New Project

Project Name

Iris Project

Project Visibility

Private - Only added collaborators can view the project.

Public - All authenticated users can view this project.

Initial Setup

Blank **Template** Local Git

Python

Templates include example code to help you get started.

Create Project

Once you've created the project, go to the project's Files page. The following files are used for the demo:

- `cdsw-build.sh` - A custom build script used for models and experiments. Pip installs our dependencies, primarily the scikit-learn library.
- `fit.py` - A model training example to be run as an experiment. Generates the `model.pkl` file that contains the fitted parameters of our model.
- `predict.py` - A sample function to be deployed as a model. Uses `model.pkl` produced by `fit.py` to make predictions about petal width.

Related Information

[Engines for Experiments and Models](#)

Train the Model

This topic shows you how to run experiments and develop a model using the `fit.py` file.

About this task

The `fit.py` script tracks metrics, mean squared error (MSE) and R2, to help compare the results of different experiments. It also writes the fitted model to a `model.pkl` file.

Procedure

1. Navigate to the Iris project's Overview Experiments page.
2. Click Run Experiment.
3. Fill out the form as follows and click Start Run. Make sure you use the Python 3 kernel.

The screenshot shows a 'Run New Experiment' dialog box with the following fields and options:

- Script:** fit.py
- Arguments:** Enter Arguments
- Engine Kernel:** Python 3 (selected), Python 2, Scala, R
- Engine Profile:** 1 vCPU / 2 GiB Memory
- Comment:** Testing a run ..

Buttons: Cancel, Start Run

4. The new experiment should now show up on the Experiments table. Click on the Run ID to go to the experiment's Overview page. The Build and Session tabs display realtime progress as the experiment builds and executes.
5. Once the experiment has completed successfully, go back to its Overview page. The tracked metrics show us that our test set had an MSE of ~ 0.0078 and an R2 of ~ 0.0493 . For the purpose of this demo, let's consider this an accurate enough model to deploy and use for predictions.

Run-21

Overview **Session** Build

Configuration

Script	fit.py
Arguments	
Comment	
Build Snapshot	cd61c8ac443de924189a55c5562d29268bbcb539
Created At	6/21/18 6:06 PM
Submitter	admin

Metrics

mean_sq_err	0.007866659505691643
r2	0.04934628330010382

Output

model.pkl

[Add to Project](#)

6. Once you have finished training and comparing metrics from different experiments, go to the experiment that generated the best model. From the experiment's Overview page, select the model.pkl file and click Add to Project.

This saves the model to the project filesystem, available on the project's Files page. We will now deploy this model as a REST API that can serve predictions.

Deploy the Model

This topic shows you how to deploy the model using the predict.py script from the Python template project.

About this task

The predict.py script contains the predict function that accepts petal length as input and uses the model built in the previous step to predict petal width.

Procedure

1. Navigate to the Iris project's Overview Models page.

2. Click New Model and fill out the fields. Make sure you use the Python 3 kernel. For example:

Create a Model

General

Name ***Description ***

Build

File ***Function *****Example Input ?**

```
{  
  "petal_length": 5.4  
}
```

Example Output ?

```
{ "result": "value" }
```

Kernel

- Python 2
 Python 3
 R

Comment

Deployment

Engine Profile**Replicas**[Set Environmental Variables](#)

3. Deploy the model.
4. Click on the model to go to its Overview page. As the model builds you can track progress on the Build page. Once deployed, you can see the replicas deployed on the Monitoring page.
5. To test the model, use the Test Model widget on the model's Overview page.

Test Model

Input

```
{  
  "petal_length": 5.4  
}
```

Test [Reset](#)

Result

Status	● success
Response	1.8826221434150965
Replica ID	predict-petal-width-2-9-7cf557b957-5wjld