

Machine Learning

Cloudera Machine Learning Runtimes

Date published: 2020-07-16

Date modified: 2024-06-11

CLOUDERA

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

Legal Notice

© Cloudera Inc. 2024. 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

Managing ML Runtimes.....	5
Adding new ML Runtimes.....	5
Adding Custom ML Runtimes through the Runtime Catalog.....	5
Adding ML Runtimes using Runtime Repo files.....	5
ML Runtimes versus Legacy Engine.....	7
Using Runtime Catalog.....	8
Disabling and Deleting Runtimes.....	11
Enabling Runtimes.....	11
Deleting Runtimes.....	11
PBJ Workbench.....	11
Dockerfile compatible with PBJ Workbench.....	14
PBJ Runtimes and Models.....	15
Example models with PBJ Runtimes.....	16
Using ML Runtimes Addons.....	17
Adding Hadoop CLI to ML Runtime Sessions.....	17
Adding Spark to ML Runtime Sessions.....	18
Turning off ML Runtimes Addons.....	18
ML Runtimes NVIDIA GPU Edition.....	18
Testing ML Runtime GPU Setup.....	19
ML Runtimes NVIDIA RAPIDS Edition.....	20
Using Editors for ML Runtimes.....	21
Using JupyterLab with ML Runtimes.....	21
Installing a Jupyter extension.....	22
Installing a Jupyter kernel.....	22
Using Conda Runtime.....	23
Installing Additional ML Runtimes Packages.....	24
Restrictions for upgrading R and Python packages.....	25
Custom Runtime Addons with CML.....	25
ML Runtimes Environment Variables.....	28
ML Runtimes Environment Variables List.....	28
Accessing Environmental Variables from Projects.....	30

Customized Runtimes.....30

- Creating Customized ML Runtimes.....31
 - Create a Dockerfile for the Custom Runtime Image..... 31
 - Metadata for Custom ML Runtimes..... 32
 - Editor Customization..... 33
 - Build the New Docker Image..... 33
 - Distribute the Image..... 33
 - Add the new ML Runtime..... 34
- Limitations..... 34
- Add Docker registry credentials and certificates..... 35

Pre-Installed Packages in ML Runtimes.....35

Managing ML Runtimes

Provides overview, installation, set up, configuration, and customization information for Machine Learning Runtimes.

ML Runtimes are responsible for running the code written by users and intermediating access to the Data Hub.

You can think of an ML Runtime as a virtual machine, customized to have all the necessary dependencies to access the computing cluster while keeping each project's environment entirely isolated. To ensure that every ML Runtime has access to the parcels and client configuration managed by the Cloudera Manager Agent, a number of folders are mounted from the host into the container environment.

ML Runtimes have been open sourced and are available in the [cloudera/ml-runtimes](https://github.com/cloudera/ml-runtimes) GitHub repository. If you need to understand your Runtime environments fully or want to build a new Runtime from scratch, you can access the Dockerfiles that were used to build the ML Runtime container images in this repository.

Adding new ML Runtimes

CML provides two ways to add new Runtimes to the Runtime Catalog.



Note: You must have system administrator permission to add a new ML Runtime. Alternatively, in Site Administration Settings Access Control, an admin can enable Allow users to register ML Runtimes. If enabled, the names of users who register ML Runtimes is displayed on the Runtime Catalog page. Users are not permitted to deprecate or disable the added runtimes.

1. Adding Custom ML Runtimes through the Runtime Catalog

Create your own Custom ML Runtime and add it through the Runtime Catalog

2. Adding ML Runtimes using Runtime Repo files

Host a file from where CML will automatically pull in new ML Runtimes and add them to the Runtime Catalog.

When adding Runtimes to CML from password protected repositories, you need to add the necessary Docker credentials to CML. See more on the *Add Docker registry credentials* page.

Related Information

[Add Docker registry credentials and certificates](#)

Adding Custom ML Runtimes through the Runtime Catalog



Note: You must have system administrator permission to add a new ML Runtime.

When you open the Runtime Catalog as an administrator, you can add a new Custom Runtime with the “Add Runtime” button. You can read more about building a Custom Runtime image and adding to the catalog in the topic *Creating Customized ML Runtimes*.

Note, that you can not add Cloudera created Runtimes through the Runtime Catalog.

Related Information

[Creating Customized ML Runtimes](#)

Adding ML Runtimes using Runtime Repo files

Runtime Repo files are JSON files that contain all the details of ML Runtimes that are needed by CML to add these ML Runtimes to the Runtime Catalog. When these files are hosted on URLs accessible to CML, these URLs can be

registered in CML. If Enable Runtime Updates is selected in Site Administration Runtime , ML Runtimes that appear in the Runtime Repo files will be automatically added to the Runtime Catalog within 24 hours. Additionally, you can click Update Runtimes now to immediately update the Runtime Catalog. Using Runtime Repo files, both Custom and Cloudera provided Runtimes can be added to CML workspaces.

To add, edit, or remove Runtime Repo files

1. Log in as administrator.
2. Navigate to Site Administration Runtime .
3. Add, edit or delete Runtime Repo files in the Runtime Updates section.

Cloudera provided Runtime Repo files

CML uses Runtime Repo files to automatically add new Runtimes to CML workspaces when Runtime updates are enabled on the workspace. Cloudera hosts its own Runtime Repo files that always contain the details of the latest released ML Runtimes and Data Visualization Runtimes. By default, you can find these Cloudera hosted Runtime Repos registered in your CML workspace:

Name: Cloudera ML Runtimes

URL: <https://archive.cloudera.com/ml-runtimes/latest/artifacts/repo-assembly.json>

Name: Cloudera DataViz Runtime

URL: <https://archive.cloudera.com/cdv/latest/artifacts/repo-assembly.json>

Self created Runtime Repo files

You can create your own Runtime Repo files and register them in CML. CML checks these Runtime Repo files for changes every 24 hours and adds any new ML Runtimes found in these files automatically to the Runtime Catalog.

To create a Repo assembly file:

1. Create a JSON file with the same structure as the Cloudera provided one:

```
{
  "assembly_metadata_version": 1,
  "runtimes": [
    {
      "image_identifier": string,
      "runtime_metadata_version": int,
      "editor": string,
      "edition": string,
      "description": string,
      "kernel": string,
      "full_version": string,
      "short_version": string,
      "maintenance_version": int,
      "git_hash": string,
      "gbn": int
    }
  ]
}
```

2. Fill in the details of one or more ML Runtimes. If you are adding Cloudera-created Runtimes, use the values from the Cloudera-provided Runtime Repo files. For Custom Runtimes, git_hash should be an empty string, and gbn should be set to zero. All other fields should be filled according to the information in *Metadata for Custom ML Runtimes*.
3. Host the JSON file on an URL that CML is able to access.
4. Add the Runtime Repo file to CML on the Site Administration Runtime page.

Related Information

[Metadata for Custom ML Runtimes](#)

ML Runtimes versus Legacy Engine

While Runtimes and the Legacy Engine are both container images that contain the Linux OS, interpreter(s), and libraries, ML Runtimes keeps the images small and improves performance, maintenance, and security.



Note: Legacy Engines are deprecated since June 2021 and will be removed in a future release. Starting with version 2.0.38, on new workspaces Legacy Engines are disabled by default and no Legacy Engine image is registered in the workspace. Cloudera recommends using ML Runtimes for all new projects, and urges customers to migrate existing Engine-based projects to ML Runtimes.

Runtimes and the Legacy Engine serve the same basic goal: they are container images that contain a complete Linux OS, interpreter(s), and libraries. They are the environment in which your code runs. However, ML Runtimes design keeps the images small, which improves performance, maintenance, and security.

There is one Legacy Engine. The Engine is monolithic. It contains the machinery necessary to run sessions using all four Engine interpreter options that Cloudera currently supports (Python 2, Python 3, R, and Scala) and a much larger set of UNIX tools including LaTeX. The Conda package manager was available in the Legacy Engine. Conda is not available in ML Runtimes.

Runtimes are the future of CML. There are many Runtimes. Currently each Runtime contains a single interpreter (for example, Python 3.8, R 4.0) and a set of UNIX tools including `gcc`. Each Runtime supports a single UI for running code (for example, the Workbench or JupyterLab).

To migrate from Legacy Engine to Runtimes, you'll need to modify your project settings. See *Modifying Project Settings* for more information.

Disable Engines

Starting with version 2.0.38, Legacy Engines are disabled by default on the workspace level. This setting is accessible in *Site Administration Runtime Disable Engines*. Select this checkbox on upgraded workspaces to disable Legacy Engines and change all existing project types to ML Runtime. Disabling Legacy Engines prohibits changing the project type to Legacy Engine.

This has the following effect for workloads using Legacy Engine kernels:

- Model builds cannot be redeployed
- Applications cannot be restarted
- Jobs cannot be started, including any kind of scheduled Jobs

To re-enable Legacy Engines go to *Site Administration Runtime* and deselect the *Disable Engines* checkbox under *Engine Images*.

Jupyter

Our Python Runtimes support JupyterLab, a general purpose IDE from the Jupyter project. The engine supports Jupyter Notebook, a simpler UI focused on Notebooks. If you prefer the simpler Notebook UI, choose *Classic Notebook* from the JupyterLab Help menu. To further customize the JupyterLab experience on CML see *Using Editors for ML Runtimes*.

Build dependencies

Runtimes generally include fewer UNIX tools than the Legacy Engine. This means you are more likely to find that you cannot install a Python or R package because the Runtime is missing a build dependency such as a library. This should not happen often with Python. Most Python packages are distributed as precompiled “wheels”, so there are no build dependencies. It is more likely to happen with R packages because precompiled packages are not available for our architecture. We have tried to cover most common use cases, but if you find you cannot build something, then please contact customer support.

Using pip to install libraries in Python

To install a Python library from within Workbench or JupyterLab we recommend you use `%pip` (for example, `%pip install sklearn`). `%pip` is a “magic” command that is guaranteed to point to the right version of pip. This is a good habit to get into, as it will work outside CML. Note you do not need to add “3” to install a Python 3 library.

If you prefer to use the pip executable directly, both pip and pip3 work. This is because Runtimes do not include Python 2. Like any shell command, precede it with “!” to run it from within Workbench or JupyterLab (for example, !pip install sklearn. In the Legacy Engine you must use pip3 to install Python 3 packages and the %pip magic command is not supported.

Python paths

Python Runtimes include preinstalled Python packages at /usr/local/lib/python/<version>/site-packages. The pre-installed packages and versions are documented in *Pre-Installed Packages in ML Runtimes*.

When you use pip, you install packages into the current project (not a runtime image) at /home/cdsd/.local/lib/python/<version>/site-packages. This means you need to reinstall packages if you change Python versions.

In most cases, you can install a newer version of a package preinstalled in /usr/local into your project. For example, we preinstall numpy and you can install a newer version. But there are some exceptions to this: if you install matplotlib, ipykernel, or its dependencies (ipython, traitlets, jupyter_client, and tornado) then you may break your ability to launch sessions.

If you accidentally install these packages (or you see unexpected behavior when you switch a project from Legacy Engine to Runtimes), the simplest solution is to delete /home/cdsd/.local/lib/python and reinstall your project's dependencies from the project overview page.

R paths

R Runtimes include preinstalled R packages at /usr/local/lib/R/library/. The pre-installed packages and versions are documented in *Pre-Installed Packages in ML Runtimes*.

When you use install.packages(), you install packages into the current project (not a runtime image) at /home/cdsd/.local/lib/R/<version>/library (for example, \$R_LIBS_USER). This means you need to reinstall packages if you change R versions.

Note the R project package path in Legacy Engines. If you use engines, you install packages to /home/cdsd/R. The change to /home/cdsd/.local/lib/R/<version>/library was made to support multiple versions of R.

In most cases, you can install a newer version of a package preinstalled /usr/local into your project. For example, we preinstall ggplot2 and you can install a newer version. But there are two exceptions to this. If you install Cairo or Rserve they may break your ability to launch sessions.

If you accidentally install these packages (or you see unexpected behavior when you switch a project from Legacy Engine to Runtimes), the simplest solution is to delete /home/cdsd/.local/lib/python and reinstall your project's dependencies from the project overview page.

Related Information

[Creating a Project with Legacy Engine Variants](#)

[Modifying Project Settings](#)

Using Runtime Catalog

You can use the Runtime catalog to list all runtimes that are available for your deployment. A Recommended label is added to some runtime variants recommended for use by Cloudera, so the administrator users can easily identify the optimal selections. Administrator users can set runtime variants as default runtime variants using the checkbox.

About this task

The Runtime Catalog lists information about each of the available runtimes. Information includes the editor and kernel supported by the runtime along with the edition, version, and a brief description.


Administrator users can use the Hide Disabled toggle so that only runtimes with Enabled or Deprecated status display. The toggle is on by default.

The Status filter also applies to runtime variants.


Procedure


Click Runtime Catalog in the left Navigation bar to display a list of all available runtimes.





 Home


ALL


 Projects


 Sessions


 Experiments


 Models


 Jobs


 Applications

 AMPs

 **Runtime Catalog**

 Learning Hub

 User Settings

 Site Administration

Runtime Catalog

 Search Runtime

Editor

<input type="checkbox"/>	Status ▴ ▾	Editor
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	JupyterLab
<input type="checkbox"/>	✓	PBJ Workbench
<input type="checkbox"/>	✓	PBJ Workbench

Disabling and Deleting Runtimes

A key feature of Cloudera Machine Learning, Runtime Catalog, is the ability to disable and enable one or more runtimes at once.

About this task

You can disable one or many runtimes at once.

Procedure

1. Select Runtime Catalog in the left-side Navigation to display all available runtimes.
2. Select one or more Runtimes.
3. Select Actions.
4. Select Set to Disabled.

Enabling Runtimes

A key feature of Cloudera Machine Learning, Runtime Catalog, is the ability to enable and disable one or more runtimes at once.

About this task

You can enable one or many runtimes at once.

Procedure

1. Select Runtime Catalog in the left-side Navigation to display all available runtimes.
2. Select one or more Runtimes.
3. Select Actions.
4. Select Set to Enabled.

Deleting Runtimes

Runtimes cannot be deleted from the Runtime Catalog. Instead, follow the steps to disable a runtime, and then ensure that the Hide Disabled toggle is set, so that disabled runtimes are not visible.

PBJ Workbench

The PBJ Workbench features the classic workbench UI backed by the open-source Jupyter protocol pre-packaged in a runtime image. Users can easily choose this runtime image when launching a session. The open-source Jupyter infrastructure eliminates the dependency on proprietary CML code when building a Docker image, allowing runtime images to be built more quickly. The PBJ Workbench enables you to construct runtime images on Ubuntu base images (including non-Cloudera base images) and use them with the CML workbench.

ML Runtimes have been open sourced and are available in the [cloudera/ml-runtimes](#) GitHub repository. If you need to understand your Runtime environments fully or want to build a new Runtime from scratch, you can access the Dockerfiles that were used to build the ML Runtime container images in this repository.

The PBJ Workbench is available by default, but you have to select it when you launch a session.

1. Click New Session.
2. In Runtime Editor, select PBJ Workbench
3. Click Start Session.

Now you can use the PBJ Workbench as you would the normal workbench.

The requirements and preparatory steps for creating a PBJ Workbench are described in each section below.

PBJ Workbench setup: Python installation

PBJ Runtimes need to have Python installed, even if the Runtime is intended to run another kernel in CML, e.g. R. The minimum Python version supported is 3.7. Python can be installed from the base image's package manager or compiled by the user.

The minimal requirements that must be satisfied by a custom PBJ Runtime image include:

- The actual Python binary or a symlink to it must exist at the following path: /usr/local/bin/python3
- The Python binary must be on the PATH under the name "python", meaning that executing the command "python" in a terminal shall start up Python.
- Executing "python --version" should result in a version greater than 3.7
- If the Runtime is configured to run a Python kernel in CML, the commands "python" and "/usr/local/bin/python3" must start up the same Python process that is registered as a Jupyter Kernel (see below).

If the method you chose to install Python does not place the Python binary under /usr/local/bin/python3, or does not create the command "python", please create appropriate symlinks.

Install Jupyter dependencies and register your kernel

First, the Jupyter Kernel Gateway, version 2.5.2 must be installed into the Docker image. This example command may need to be modified depending on the filename and path of the pip executable in the image.

```
RUN pip3 install "jupyter-kernel-gateway==2.5.2"
```

The path to the Jupyter executable installed by pip should be noted, and the command to run Jupyter Kernel Gateway must be incorporated into the ML_RUNTIME_JUPYTER_KERNEL_GATEWAY_CMD environment variable in the Docker image:

```
ENV ML_RUNTIME_JUPYTER_KERNEL_GATEWAY_CMD="/path/to/jupyter kernelgateway"
```



Note: The highest supported version of the jupyter-client library is version 7.4.9. Do not upgrade to version 8 or higher.

When launching the Runtime in CML, the correct IP address - port configuration for Jupyter Kernel Gateway is set automatically by CML.

Next, a Jupyter kernel has to be registered. Each instance of the PBJ Workbench communicates with the Jupyter kernel installed in the Runtime via the Jupyter protocol. Kernels are available for a wide variety of languages and versions. Install the kernel of your choice to the image by following its installation instructions. A kernel named python3 is registered by default when installing jupyter-kernel-gateway via pip. Installed Jupyter kernels can be listed by running the following command in a container created from the image:

```
path/to/jupyter kernelspec list
```

The name of your chosen kernel must be incorporated into the ML_RUNTIME_JUPYTER_KERNEL_NAME environment variable in the Docker image. For example, if your kernel's name is python3, the following must be included in the Dockerfile:

```
ENV ML_RUNTIME_JUPYTER_KERNEL_NAME=python3
```

Add the cdsw user

User code will be run in the image under user and group 8536:8536 . Associate these id's with the name cdsw in the image by adding the following command to the dockerfile:

```
RUN groupadd --gid 8536 cdsw && \  
    useradd -c "CDSW User" --uid 8536 -g cdsw -m -s /bin/bash cdsw
```

Relax permissions so that Cloudera client config can be written

All code in the runtime container, including initial setup, will be run as the cdsw user. The initial setup includes linking client files for Cloudera data services out to their standard paths. To facilitate this, permissions to the following paths should be set so that user 8536 can write to them and to their subfolders:

- /etc
- /bin
- /usr/share/java
- /opt
- /usr

Also set the permissions of the following folders and all their subfolders to 777.

- /etc
- /etc/alternatives

Additional requirements

- ML_RUNTIME_METADATA_VERSION environment variable and the corresponding Docker label must be set to 2.
- To use the PBJ Workbench editor, the ML_RUNTIME_EDITOR environment variable and the corresponding Docker label must be set to "PBJ Workbench". If using a 3rd party editor (for example, JupyterLab or RStudio), set the ML_RUNTIME_EDITOR environment variable and Docker label to the desired value. Note that "Workbench" and "PBJ Workbench" are reserved names.
- Base image must be Ubuntu.
- Bash must be installed and must be configured as the default terminal used by the cdsw user.
- In case the PBJ Runtime is running the R kernel, the kernel must be registered with the IRkernel package.
- The executable that is registered as a Jupyter kernel must be on PATH, must be found by the "which" command and must be named after the programming language of the kernel. E.g. the name of the executable must be:
 - `python` in case of a Python kernel (python3 is not sufficient)
 - `R` in case of an R kernel, etc.
- In the case of using a virtual / conda environment and a Python kernel, we recommend configuring PATH such that the default "pip" command corresponds to the python executable registered as a Jupyter kernel.
- CML mounts the project's filesystem under the path /home/cdsw and erases any file installed to that path in the Runtime image. Therefore, custom Runtime images should not install anything under the home folder of the cdsw user.
- On the other hand, once the Runtime image starts up in CML, the kernel must be configured to install new packages to user site libraries under /home/cdsw. That way, newly installed packages will persist in the project's filesystem.
- The package xz-utils must be installed on the Runtime image.
- The following binaries should be reachable on the PATH: kinit, klist, ktutil, and sshd. These are installed on Ubuntu as part of the following packages: krb5-user and ssh.

Dockerfile compatible with PBJ Workbench

This Dockerfile produces a runtime image that is compatible with the PBJ Workbench from a third-party base image.



Note: Apple silicon users should change the first line in the script to the following:

FROM --platform=linux/amd64 ubuntu:22.04

```
FROM ubuntu:22.04
USER root
# Install Python
# Note that the package python-is-python3 will alias python3 as python
RUN apt-get update && apt-get install -y --no-install-recommends \
    krb5-user python3.10 python3-pip python-is-python3 ssh xz-utils

# Configure pip to install packages under /usr/local
# when building the Runtime image
RUN pip3 config set install.user false

# Install the Jupyter kernel gateway.
# The IPython kernel is automatically installed
# under the name python3,
# so below we set the kernel name to python3.
RUN pip3 install "jupyter-kernel-gateway==2.5.2"

# Associate uid and gid 8536 with username cdsw
RUN \
    addgroup --gid 8536 cdsw && \
    adduser --disabled-password --gecos "CDSW User" --uid 8536 --gid 8536 cdsw

# Set up Python symlink to /usr/local/bin/python3
RUN ln -s $(which python) /usr/local/bin/python3

# Install any additional packages.
# apt-get install ...
# pip install ...

# configure pip to install packages to /home/cds
# once the Runtime image is loaded into CML
# do not install Python packages in the Dockerfile after this line
RUN /bin/bash -c "echo -e '[install]\nuser = true' > /etc/pip.conf"

# Relax permissions to facilitate installation of Cloudera
# client files at startup
RUN for i in /bin /opt /usr /usr/share/java; do \
    mkdir -p ${i}; \
    chown cds ${i}; \
    chmod +rw ${i}; \
    for subfolder in `find ${i} -type d` ; do \
        chown cds ${subfolder}; \
        chmod +rw ${subfolder}; \
    done \
done

RUN for i in /etc /etc/alternatives; do \
    mkdir -p ${i}; \
    chmod 777 ${i}; \
done

# Set Runtime label and environment variables metadata
#ML_RUNTIME_EDITOR and ML_RUNTIME_METADATA_VERSION must not be changed.
```

```

ENV ML_RUNTIME_EDITOR="PBJ Workbench" \
ML_RUNTIME_METADATA_VERSION="2" \
ML_RUNTIME_KERNEL="Python 3.10" \
ML_RUNTIME_EDITION="Custom Edition" \
ML_RUNTIME_SHORT_VERSION="1.0" \
ML_RUNTIME_MAINTENANCE_VERSION="1" \
ML_RUNTIME_JUPYTER_KERNEL_GATEWAY_CMD="/usr/local/bin/jupyter kernelg
ateway" \
ML_RUNTIME_JUPYTER_KERNEL_NAME="python3" \
ML_RUNTIME_DESCRIPTION="My first Custom PBJ Runtime"

ENV ML_RUNTIME_FULL_VERSION="$ML_RUNTIME_SHORT_VERSION.$ML_RUNTIME_MAINTEN
ANCE_VERSION"

LABEL com.cloudera.ml.runtime.editor=$ML_RUNTIME_EDITOR \
com.cloudera.ml.runtime.kernel=$ML_RUNTIME_KERNEL \
com.cloudera.ml.runtime.edition=$ML_RUNTIME_EDITION \
com.cloudera.ml.runtime.full-version=$ML_RUNTIME_FULL_VERSION \
com.cloudera.ml.runtime.short-version=$ML_RUNTIME_SHORT_VERSION \
com.cloudera.ml.runtime.maintenance-version=$ML_RUNTIME_MAINTENANCE_
VERSION \
com.cloudera.ml.runtime.description=$ML_RUNTIME_DESCRIPTION \
com.cloudera.ml.runtime.runtime-metadata-version=$ML_RUNTIME_METADAT
A_VERSION

```

PBJ Runtimes and Models

The PBJ (Powered by Jupyter) Runtime enables a wide variety of language kernels to be run as CML workloads. Model workloads are currently only supported for Python and R kernels.

A new library, `cml`, is added to Runtime based Python and R workloads automatically and includes mostly the same functionality as the old `cdsw` library.

In non-PBJ Runtimes, you could point to a Python or R function and run it as a model. PBJ Runtimes open up the possibilities for users to create their own Runtimes and places few restrictions on how to do that. PBJ technology utilizes the Jupyter kernelgateway to communicate with language kernels and aims to be language independent in the future.

Migrating from the `cdsw` to `cml` library

PBJ Runtimes currently includes two supported languages, Python and R. There are new function wrappers or decorators, described below, that must be used in order to allow previously written model code to function in a PBJ Runtime. It is important to note that these support decorators or wrappers are transparent, that is, they have no effect, in non-PBJ workloads and non-model workloads.

Table 1: Python names

CDSW	CML
<code>call_model</code>	<code>models_v1.call_model</code>
-	<code>models_v1.cml_model</code>
<code>get_auth</code>	<code>utils_v1.get_auth</code>
<code>track_file</code>	Removed
<code>model_metrics</code>	Replaced by <code>models_v1.cml_model(metrics=True)</code>
<code>read_metrics</code>	<code>metrics_v1.read_metrics</code>
<code>track_metric</code>	<code>metrics_v1.track_metric</code>

CDSW	CML
track_delayed_metrics	metrics_v1.track_delayed_metrics
track_aggregate_metrics	metrics_v1.track_aggregate_metrics
launch_workers	workers_v1.launch_workers
list_workers	workers_v1.list_workers
stop_workers	workers_v1.stop_workers
await_workers	workers_v1.await_workers

Table 2: R names

CDSW	CML
html	Planned for future release
iframe	Planned for future release
markdown	Planned for future release
display.help	Planned for future release
code	Planned for future release
image	Planned for future release
get.auth	get.auth
launch.workers	launch.workers
stop.workers	stop.workers
list.workers	list.workers
await.workers	await.workers
track.metric	Removed
track.file	Removed
-	cml_model

Library availability

In CML, The cdsw and s libraries are available in different runtime and engine types.

Runtime / Engine type	Library availability
Legacy Engine	cdsw
Classic Runtime	cdsw, cml
PBJ Runtime	cml



Note: The cml library is now used by default, instead of the cdsw library. However, if a project was created on a previous CML release, it might use the deprecated cdsw library. Code using the cdsw library will not run on a PBJ Runtime. Code running on PBJ Runtimes must use the cml library.

Example models with PBJ Runtimes

The library cml includes the package, models_v1. This package includes the cml_model decorator that can be used to allow a function to work as a model in a PBJ Runtime. It can also be used to enable gathering of model metrics.

The package has an optional boolean argument, called metrics, that when set to True, enables the model_metric decorator functionality from the deprecated cdsw lib. The default value of the metrics parameter is False. Therefore,

this example does not have model metric gathering enabled, but the R example does. In both examples the function to be used for the model is called predict.

Python example

Example #1

```
import cml.models_v1 as models

@models.cml_model
def predict(args):
    return args["x"]*2
```

Example #2:

```
import cml.models_v1 as models

@models.cml_model(metrics=True)
def predict(args):
    return args["x"]*2
```

R example

Almost all of the functionality in the cdsw library for R has also been migrated to the cml library and is available automatically for every R Runtime workload. It includes a new function wrapper, called cml_model, to be used for model entry point functions in PBJ Runtimes. The function to be used for the model in this case is called predict.

```
library(cml)

predict <- cml_model(function(args) {
  return(args$x*2)
})
```

Using ML Runtimes Addons

ML Runtime Addons allow you to add Spark and Hadoop CLI to sessions run on projects using ML Runtime images.

While Legacy Engines include support for both Spark and Hadoop CLI, ML Runtimes do not contain Spark and Hadoop CLI binaries to keep them small and lean. Instead Spark and Hadoop binaries are stored in persistent storage on an NFS server and can be added to your ML Runtime sessions.

Adding Hadoop CLI to ML Runtime Sessions

Hadoop CLI can be enabled only on sessions that are selected to use Spark.

About this task

To add Hadoop CLI to sessions run on projects using ML Runtimes images:

Procedure

1. You can view all available ML Runtime addons by selecting the Site Administration>Runtime/Engine tab and viewing Runtime Addons.

2. To include Hadoop in all sessions created in the workspace, under Site Administration>Runtime/Engine, choose the desired Hadoop version in the pull down menu next to Hadoop CLI Version.



Note: Hadoop CLI can be enabled only on sessions that are selected to use Spark.

This will add Hadoop CLI to all sessions created in the workspace.

3. To use Hadoop commands for your session, click the Terminal Access button at the top of the Session window. CML launches a terminal window in which you can use Hadoop commands.

Adding Spark to ML Runtime Sessions

You can add Spark to ML Runtime Sessions using the ML Runtimes Addons. Both Spark and Hadoop CLI are enabled when you enable Spark.

About this task

To add Spark to sessions run on projects using ML Runtimes images:

Procedure

1. You can view all available ML Runtime addons by selecting the Site Administration>Runtime/Engine tab and viewing Runtime Addons.
2. Start a New Session for an ML Runtimes project.
3. Click the Enable Spark option, then select the Spark version.
4. Click Start Session.
5. You can now run a Spark job for your session.
The Logs tab displays the executors in your Spark job.
6. After you start a Spark job, you can access the Spark UI by clicking the Spark UI button at the top of the Session window.

Turning off ML Runtimes Addons

ML Runtimes Addons is turned on by default. However, you can disable ML Runtimes Addons.

Procedure

1. Select Site Administration > Settings.
2. Under Feature Flags, uncheck the checkbox next to Allow users to Run ML Runtimes Addons.

ML Runtimes NVIDIA GPU Edition

The NVIDIA GPU Edition Runtimes are built on top of NVIDIA CUDA docker images. CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers can dramatically speed up computing applications by harnessing the power of GPUs.

Cloudera Machine Learning supports GPUs using particular ML Runtime Editions. NVIDIA GPUs together with related specialized software can be utilized using the NVIDIA GPU Edition.

Version information:

Version	Additional software	Tested OS	Tested Driver Version
2021.09	CUDA 11.4	RHEL 7.6	NVIDIA driver 460.56

Compatibility information:

Runtime Version	CUDA Version	Kernels	Editors	Base Image
2021.02	CUDA 11.1	Python 3.7 and Python 3.8	Workbench and JupyterLab	

ML Runtimes inherit the compatibility requirements for NVIDIA CUDA. For compatibility information, visit the following pages:

- [NVIDIA/CUDA compatibility matrix](#)
- [Tensorflow requirements](#)
- [Pytorch requirements](#)

Testing ML Runtime GPU Setup

You can use the following simple examples to test whether the new ML Runtime is able to leverage GPUs as expected.

1. Go to a project that is using the ML Runtimes NVIDIA GPU edition and click Open Workbench.
2. Launch a new session with GPUs.
3. Run the following command in the workbench command prompt to verify that the driver was installed correctly:

```
! /usr/bin/nvidia-smi
```

4. Use any of the following code samples to confirm that the new engine works with common deep learning libraries.

Pytorch

```
!pip3 install torch==1.4.0
from torch import cuda
assert cuda.is_available()
assert cuda.device_count() > 0
print(cuda.get_device_name(cuda.current_device()))
```

Tensorflow

```
!pip3 install tensorflow-gpu==2.1.0
from tensorflow.python.client import device_lib
assert 'GPU' in str(device_lib.list_local_devices())
device_lib.list_local_devices()
```

Keras

```
!pip3 install keras
from keras import backend
assert len(backend.tensorflow_backend._get_available_gpus()) > 0
print(backend.tensorflow_backend._get_available_gpus())
```

ML Runtimes NVIDIA RAPIDS Edition

The RAPIDS Edition Runtimes are built on top of community built RAPIDS docker images. The RAPIDS suite of software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs. It relies on NVIDIA CUDA primitives for low-level compute optimization, but exposes that GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces.



Note: RAPIDS require NVIDIA Pascal or better GPUs. You need to use P3 or newer EC2 instances on AWS to meet this requirement.

Visit rapids.ai for more information.



Note: RAPIDS on Spark is not support as Spark3 is not supported on CDSW.

ML Runtimes RAPIDS edition differs from other Runtime Editions in the following ways:

- Python maintenance versions differ from what is being used in the Standard and NVIDIA GPU edition runtimes. These Python kernels are coming from the RAPIDS base image.
- Pre-installed Python packages and package versions differ from what's in Standard and NVIDIA GPU edition runtimes.

RAPIDS Runtimes are an optional extension of the Cloudera ML Runtime distribution. The RAPIDS images are not distributed automatically. Administrators can register them in the Runtime Catalog. The following RAPIDS editions are available for the 2021.04 Runtime version:

RAPIDS and CUDA Version	Kernel	Editor	Base Image	Docker
RAPIDS 0.18 CUDA 11.0	Python 3.7	Workbench editor	rapidsai/rapidsai-core:0.18-cuda11.0-base-ubuntu20.04-py3.7	docker.repository.cloudera.com/cloud- era/cdsw/ml-runtime- workbench-python 3.7-rapids:2021.12.1- b17
RAPIDS 0.18 CUDA 11.0	Python 3.8	Workbench editor	rapidsai/rapidsai-core:0.18-cuda11.0-base-ubuntu20.04-py3.8	docker.repository.cloudera.com/cloud- era/cdsw/ml-runtime- workbench-python 3.8-rapids:2021.12.1- b17
RAPIDS 0.18 CUDA 11.0	Python 3.7	JupyterLab editor	rapidsai/rapidsai-core:0.18-cuda11.0-base-ubuntu20.04-py3.7 Note: This image is not based on NVIDIA's JupyterLab installation so RAPIDS library examples are not installed.	docker.repository.cloudera.com/cloud- era/cdsw/ml-runtime- jupyterlab-python3.7- rapids:2021.12.1-b17

RAPIDS and CUDA Version	Kernel	Editor	Base Image	Docker
RAPIDS 0.18 CUDA 11.0	Python 3.8	JupyterLab editor	rapidsai/rapidsai-core:0.18-cuda11.0-base-ubuntu20.04-py3.8 Note: This image is not based on NVIDIA's JupyterLab installation so RAPIDS library examples are not installed.	docker.repository.cloudera.com/cloud-dera/cdsw/ml-runtime-jupyterlab-python3.8-rapids:2021.12.1-b17

RAPIDS Python environment

The RAPIDS python libraries are distributed and present as a conda environment in the docker image. While it's possible to install additional libraries for your project via conda, these packages will be installed to a non-persistent part of the file system and won't persist between sessions and jobs.

Installing additional libraries via pip is supported and works the same way as in other Runtimes.



Note: The RAPIDS images have the official Anaconda channels configured for conda, if you decide to use conda to install packages, make sure you are familiar with Anaconda's license requirements.

Related Information

[Testing ML Runtime GPU Setup](#)

Using Editors for ML Runtimes

Cloudera Machine Learning provides a Workbench UI to edit and run code, but we also provide a preconfigured JupyterLab runtime to allow this. Choose the editor you prefer when launching a ML Runtime session.

Using JupyterLab with ML Runtimes

JupyterLab is a web-based interactive development environment for Jupyter notebooks, code, and data.

You can use JupyterLab to configure and arrange the user interface to support a wide range of workflows in data science, scientific computing, and machine learning. JupyterLab allows you to use extensions that add new components and integrate with existing ones.

You must install any files necessary for JupyterLab, including configuration, customization, extensions, and kernels, into /home/cdsw for each project.

Code Completion in JupyterLab

You can use JupyterLab to write and run code in notebooks or in a traditional .py file. Completion works out of the box in notebooks and consoles. If you are working in a .py file in the text editor, you must create a console to provide the completion suggestions. To do this:

1. Open the .py file.
2. Right click on the document and choose "Create console for editor".
3. Choose the option appropriate to the type of completion you want to perform.

For example, to get completions on "math", you must first "import math" in the console.

Set JupyterLab Timeouts

You can control JupyterLab timeouts with the following environment variables:

```
JUPYTER_KERNEL_TIMEOUT_SECONDS  
JUPYTER_SERVER_TIMEOUT_SECONDS
```

Installing a Jupyter extension

Extensions modify the appearance or behaviour of Jupyter applications (including JupyterLab and Jupyter Notebook).

Before you begin

You must install any files necessary for JupyterLab, including configuration, customization, extensions, and kernels, into /home/cdsd for each project.



Note: Any Jupyter Extension that is not specifically designed for JupyterLab should work fine with Jupyter Notebook, which is preinstalled in Legacy Engine engine:14.

About this task

The following example installs the ipython-sql extension. This extension adds a %sql magic command that allows you to communicate directly with any database supported by SQLAlchemy.

Procedure

1. Launch a JupyterLab session.
2. Open a terminal and run the following command:

```
pip install ipython-sql
```



Note: The previous command is only an example for installing ipython-sql.

3. Open a notebook or console and run the following command:

```
%load_ext sql
```

Results

You can now use the %sql magic command to connect to a particular database, and the %%sql magic command to create a cell containing an SQL query to run against that database. See the JupyterLab documentation for additional information on Jupyter magic commands.

Installing a Jupyter kernel

Jupyter kernels are “connections” through which a notebook (or other part of JupyterLab) can talk to a particular interpreter. CDSW includes one kernel in each JupyterLab Runtime: Python 3.6, Python 3.7, or Python 3.8.

Jupyter kernels are “connections” through which a notebook (or other part of JupyterLab) can talk to an interpreter. ML Runtimes come with a Jupyter kernel for Python 3.6, 3.7 or 3.8 preinstalled, but you may also install Jupyter kernels for bash or JavaScript/TypeScript.

bash

The following example describes how to enable a bash kernel in a particular project.

1. Launch a JupyterLab session.

2. Open a terminal window and run the following command:

```
pip install bash_kernel
python -m bash_kernel.install
```

You should now have the option to launch bash notebooks and consoles.

JavaScript and TypeScript

The following example describes how to enable JavaScript and TypeScript kernels in a particular project. .

1. Customize your PATH environment variable:
 - a. Navigate to Project Settings/Advanced.
 - b. Set PATH to \$HOME/node_modules/.bin:\$PATH.
2. Launch a JupyterLab session.
3. Open a terminal window and run the following command:

```
npm install tslab
tslab install
```

You should now have the option to launch JavaScript and TypeScript notebooks and console.

Using Conda Runtime

Users now can create their own Python or R Conda environments within their CML Projects that they can use in the JupyterLab editor.

Environments, installed packages, and kernel specifications are persisted on the Project file system. An example flow for creating a new Python 3.10 Conda environment is the following:

```
conda create --name myenv python=3.10
conda activate myenv
conda install ipykernel
ipython kernel install --user --name=myenv
```

JupyterLab Conda Tech Preview Runtime

You might run into some known issues while using JupyterLab Conda Runtime.

Sessions

When starting a Notebook or a Console for a specific environment, the installed packages will be available and the interpreter used to evaluate the contents of the Notebook or Console will be the one installed in the environment. However, the Conda environment is not "activated" in these sessions, therefore commands like `!which python` will return with the base Python 3.10 interpreter on the Runtime. The recommended ways to modify a Conda environments or install packages are the following:

- conda commands must be used with the `-n` or `--name` argument to specify the environment, for example `conda -n myenv install pandas`
- When installing packages with pip, use the `%pip` magic to install packages in the active kernel's environment, for example `%pip install pandas`

Applications and Jobs

To start an Application or Job, first create a launcher Python script containing the following line: `! source activate <conda_env_name> && python <job / application script.py>`

When starting the Application or Job, select the launcher script as the "Script".

Models

Models are currently not supported for the Conda Runtime.

Spark

Spark is not supported in JupyterLab Notebooks and Consoles.

Spark workloads are supported in activated Conda environments in JupyterLab Terminals, or in Jobs or Applications.

The CDSW libraries for Python and R are not available for the Conda Runtimes.

Installing Additional ML Runtimes Packages

Cloudera Machine Learning Runtimes are preloaded with a few common packages and libraries for Python. However, a key feature of Cloudera Machine Learning is the ability of different projects to install and use libraries pinned to specific versions, just as you would on your local computer.

Before you begin

Before downloading or using third-party content, you are responsible for reviewing and complying with any applicable license terms and making sure that they are acceptable for your use case.

About this task

Generally, Cloudera recommends you install all required packages locally into your project. This will ensure you have the exact versions you want and that these libraries will not be upgraded when Cloudera upgrades the ML Runtimes image. You only need to install libraries and packages once per project. From then on, they are available to any new ML Runtimes you spawn throughout the lifetime of the project.

You can install additional libraries and packages from the workbench, using either the command prompt or the terminal.



Note:

Cloudera Machine Learning does not currently support installation of packages that require root access to the hosts. For such use-cases, you will need to create a new custom runtime that extends the base image with the required packages. For instructions, see [Customized Runtimes](#) on page 30.

About this task

(Python) Install Packages Using Workbench Command Prompt

To install a package from the command prompt:

1. Navigate to your project's Overview page. Click New Session and launch a session.
2. At the command prompt (see Native Workbench Console and Editor) in the bottom right, enter the command to install the package. Some examples using Python have been provided.

Python 3

```
# Installing from console using ! shell operator and pip3:  
!pip3 install beautifulsoup4  
# Installing from terminal  
pip3 install beautifulsoup4
```

About this task

(Python Only) Using a Requirements File

For a Python project, you can specify a list of the packages you want in a requirements.txt file that lives in your project. The packages can be installed all at once using pip/pip3.

1. Create a new file called requirements.txt file within your project:

```
beautifulsoup4==4.6.0
seaborn==0.7.1
```

2. To install the packages in a Python 3 ML Runtimes, run the following command in the workbench command prompt.

```
!pip3 install -r requirements.txt
```

Restrictions for upgrading R and Python packages

Some R and Python packages should not be upgraded because doing so will break the Workbench UI.

For R Runtimes

Do not upgrade the Cairo or RServe packages in your projects using R Runtimes. If you do, you may be unable to launch sessions, jobs, experiments, or applications.

If these packages are upgraded, you will need to start a new project or to delete the directory .local/lib using the CDSW/CML Files UI ("show hidden").

For Python Runtimes

Do not upgrade the ipykernel package or its dependencies (ipython, traitlets, jupyter_client, and tornado) in your project using Python Runtimes. If you do, you may be unable to launch sessions, jobs, experiments, or applications.

If these packages are upgraded, you will need to start a new project or to delete the directory .local/lib using the CDSW/CML Files UI ("show hidden").

Python Runtimes in CML fail to import the setuptools Python library and can fail installing some Python packages

Python Runtimes in CML fail to import the setuptools Python library and therefore can fail installing some Python packages when the library setuptools is present on the Runtime or is installed into the CML project with version 60.0.0 or higher.

Python 3.10 Runtimes from the 2023.05 Runtime release ship with a newer version of setuptools, so customers can run into this issue when they are using that Runtime. Also they can run into this issue when they are using Custom Runtimes that has a newer setuptools library version or when they install a new setuptools version into their project (regardless of what Runtime they use).

Workaround: Set the environmental variable SETUPTOOLS_USE_DISTUTILS=stdlib either on a project level under Project Settings -> Advanced or on a workspace level under Site Administration -> Runtime -> Environment variables.

Custom Runtime Addons with CML

Custom ML Runtimes enable you to create runtimes with your own choice of libraries and applications, but it is also possible to further customize existing ML runtimes with additional configuration files or binaries such as connection drivers, without the effort of creating a new custom ML runtime.



Warning: Using unsecured or unlicensed libraries can cause security threats. It is your responsibility to diligently add the custom ML Runtime addons. For more information, see *Cloudera's Shared Responsibility Model*.

How this feature works

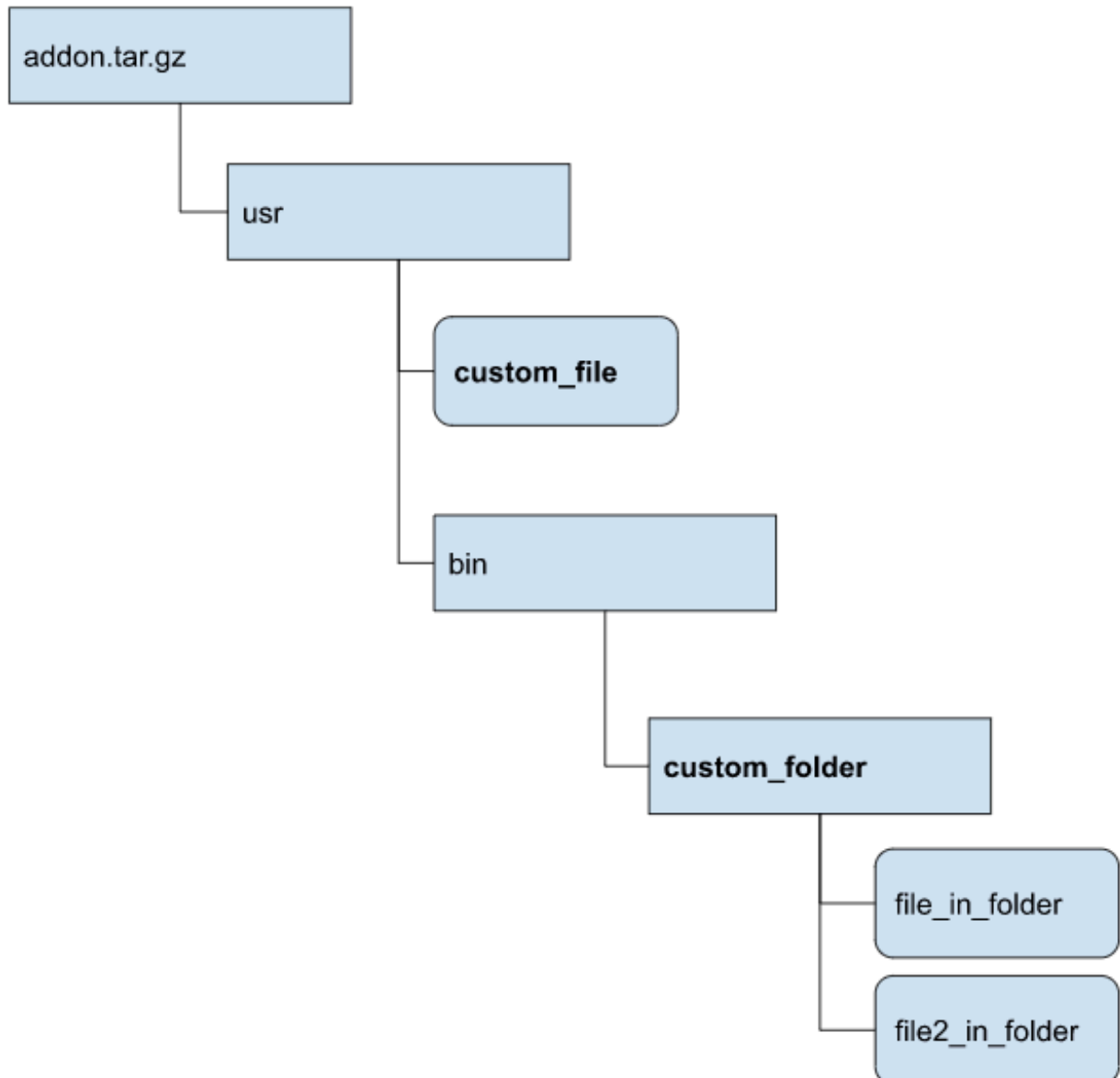
Package your dependencies into a tarball file, following the specified format. A site administrator can then register the custom runtime addon by uploading the tarball via an API call on the command line.

After a custom runtime addon has been registered, the site administrator can enable or disable addons in the UI.

When an addon is enabled, it is mounted to all newly started runtime-based workloads, in read-only mode.

Creating the Custom Runtime Addon

1. You must create a structure of files and folders that follows the pattern shown here, and package it as a tar.gz archive. For example, to mount a folder called `custom_folder`, with all of its content under `/usr/bin/custom_folder`, and `custom_file` at `/usr/custom_file`, the internal structure of the tarball should look like this:



2. Next, you create a json file that describes the custom runtime addon to register in CML. The json file should follow this structure:

```
{
  "name" : "name-of-the-addon",
  "spec" : {"paths" : ["/usr/custom_file", "/usr/bin/custom_folder"]}
```

```
}
```

3. Finally, the json file and tar.gz file are uploaded by a Site administrator. Use the following curl command, with these substitutions:

- metadata.json: the name of the json file
- addon.tar.gz: the name of the tar.gz file

```
curl -v -L '<cml_cluster>/api/v2/runtimeaddons/custom' -F metadata=@metadata.json
-F tarball=@addon.tar.gz -H 'Authorization: Bearer <APIv2 key>'
```

After the runtime add-on has been uploaded, it will be enabled and mounted to all newly started Runtime-based workloads.

Requirements

1. Name of the addon must contain only lowercase letters, numbers, underscores and hyphens. Also, it must be no longer than 35 characters. When added to CML, the addon will be prefixed with the term "custom-addon-".
2. There should be at least one path mentioned in the uploaded json file.
3. Paths in the metadata file should not end with a forward slash (/) character.
4. Requested paths will be symlinked in the workloads' filesystem by the "cdsw" user, therefore these locations must not exist in the workload's file system and "cdsw" user must have write access to these locations.

Limitations

1. Injecting Python or R libraries into workloads via Custom Runtime Addons is not supported. To accomplish this, create a Custom Runtime instead. For more information, see *Creating Customized ML Runtimes*.
2. Misconfigured Custom Runtime Addons can break user workloads (e.g. if metadata refers to paths that do not exist in the uploaded tarball archive).
3. It is not possible to add Custom Runtime Addons through the UI.
4. Custom Runtime Addons can not be updated or removed, only disabled.
5. Custom Runtime Addons are mounted to workloads in read only mode.

Troubleshooting

When a Custom Runtime Addon tries to mount in a file or folder that falls into one of the below categories, workloads might fail to come up. In that case, CML UI will show the error messages "Failed setting up runtime addons", also "Engine exited with status 33". The exact error message can help determine what scenario causes the issue.

Possible root causes CML not being able to mount in files or folders from Custom Runtime Addons:

The file or folder does not exist in the uploaded tarball or is not located under the proper path within the tarball archive.

Solution: Adjust paths in the tarball/metadata file according to the Requirements.

The file or folder a Custom Runtime Addon tries to mount exists already in the filesystem. Either the file/folder exists on the Runtime image or is mounted in by another Runtime Addon.

Solution: Due to a limitation, Custom Runtime Addons can not overwrite existing files. Customers should change the path of the mounted files.

cdsw user has no write access in the pod's file system to create symlinks to mounted files.

Solution: Change the path of the mounted files to a location that is writable by the cdsw user. Alternatively, create a Custom Runtime image with adjusted permissions and use that in all workload

Related Information

[Creating Customized ML Runtimes](#)

ML Runtimes Environment Variables

This topic describes how ML Runtimes environmental variables work. It also lists the different scopes at which they can be set and the order of precedence that will be followed in case of conflicts.

ML Runtimes environment variables behave the same way for Legacy Engines and ML Runtimes.

Environmental variables allow you to customize ML Runtimes environments for projects. For example, if you need to configure a particular timezone for a project, or increase the length of the session/job timeout windows, you can use environmental variables to do so.

CML allows you to define environmental variables for the following scopes:

Global

A site administrator for your CML deployment can set environmental variables on a global level. These values will apply to every project on the deployment.

To set global environmental variables, go to [Admin Runtime/Engine](#).

Project

Project administrators can set project-specific environmental variables to customize the ML Runtimes launched for a project. Variables set here will override the global values set in the site administration panel.

To set environmental variables for a project, go to the project's Overview page and click [Settings Advanced](#).

Job

Environments for individual jobs within a project can be customized while creating the job. Variables set per-job will override the project-level and global settings.

To set environmental variables for a job, go to the job's Overview page and click [Settings Set Environmental Variables](#).

Experiments

ML Runtimes created for execution of experiments are completely isolated from the project. However, these ML Runtimes inherit values from environmental variables set at the project-level and/or global level. Variables set at the project-level will override the global values set in the site administration panel.

Models


Model environments are completely isolated from the project. Environmental variables for these ML Runtimes can be configured during the build stage of the model deployment process. Models will also inherit any environment variables set at the project and global level. However, variables set per-model build will override other settings.

ML Runtimes Environment Variables List

The following table lists Cloudera Machine Learning environment variables that you can use to customize your project environments. These can be set either as a site administrator or within the scope of a project or a job.

Environment Variable	Description
MAX_TEXT_LENGTH	Maximum number of characters that can be displayed in a single text cell. By default, this value is set to 800,000 and any more characters will be truncated. Default: 800,000
SESSION_MAXIMUM_MINUTES	Maximum number of minutes a session can run before it times out. Default: 60*24*7 minutes (7 days) Maximum Value: 35,000 minutes
JOB_MAXIMUM_MINUTES	Maximum number of minutes a job can run before it times out. Default: 60*24*7 minutes (7 days) Maximum Value: 35,000 minutes
IDLE_MAXIMUM_MINUTES	Maximum number of minutes a session can remain idle before it exits. Default: 60 minutes Maximum Value: 35,000 minutes Idle timeouts for sessions vary by workbench type (runtime). <ul style="list-style-type: none"> Standard Workbench: Sessions timeout regardless of activity in the browser or terminal. PBJ Workbench: Sessions timeout if there is no browser activity and no terminal window is open. If a terminal window is open, the session will not timeout, regardless of whether there is activity in the terminal window. Jupyterlab: Sessions timeout if there is no browser activity. Terminal window activity is not considered. Custom runtimes: No idle timeout behavior is enforced on custom or third-party workbenches.

Per-Engine Environmental Variables: In addition to the previous table, there are some more built-in environmental variables that are set by the Cloudera Machine Learning application itself and do not need to be modified by users. These variables are set per-engine launched by Cloudera Machine Learning and only apply within the scope of each engine.

Environment Variable	Description
CDSW_PROJECT	The project to which this engine belongs.
CDSW_ENGINE_ID	The ID of this engine. For sessions, this appears in your browser's URL bar.
CDSW_MASTER_ID	If this engine is a worker, this is the CDSW_ENGINE_ID of its master.
CDSW_MASTER_IP	If this engine is a worker, this is the IP address of its master.
CDSW_PUBLIC_PORT	 <p>Note: This property is deprecated. See CDSW_APP_PORT and CDSW_READONLY_PORT for alternatives.</p> <p>A port on which you can expose HTTP services in the engine to browsers. HTTP services that bind CDSW_PUBLIC_PORT will be available in browsers at: <code>http(s)://<CDSW_ENGINE_ID>.<CDSW_DOMAIN></code>. By default, CDSW_PUBLIC_PORT is set to 8080.</p> <p>A direct link to these web services will be available from the grid icon in the upper right corner of the Cloudera Machine Learning web application, as long as the job or session is still running. For more details, see <i>Accessing Web User Interfaces from Cloudera Machine Learning</i>.</p> <p>In Cloudera Machine Learning, setting CDSW_PUBLIC_PORT to a non-default port number is not supported.</p>

Environment Variable	Description
CDSW_APP_PORT	<p>A port on which you can expose HTTP services in the engine to browsers. HTTP services that bind CDSW_APP_PORT will be available in browsers at: <code>http(s)://<\$CDSW_ENGINE_ID>.<\$CDSW_DOMAIN></code>. Use this port for applications that grant some control to the project, such as access to the session or terminal.</p> <p>A direct link to these web services will be available from the grid icon in the upper right corner of the Cloudera Machine Learning web application as long as the job or session runs. Even if the web UI does not have authentication, only Contributors and those with more access to the project can access it. For more details, see <i>Accessing Web User Interfaces from Cloudera Machine Learning</i>.</p> <p>Note that if the Site Administrator has enabled Allow only session creators to run commands on active sessions, then the UI is only available to the session creator. Other users will not be able to access it.</p> <p>Use 127.0.0.1 as the IP.</p>
CDSW_READONLY_PORT	<p>A port on which you can expose HTTP services in the engine to browsers. HTTP services that bind CDSW_READONLY_PORT will be available in browsers at: <code>http(s)://<\$CDSW_ENGINE_ID>.<\$CDSW_DOMAIN></code>. Use this port for applications that grant read-only access to project results.</p> <p>A direct link to these web services will be available to users with from the grid icon in the upper right corner of the Cloudera Machine Learning web application as long as the job or session runs. Even if the web UI does not have authentication, Viewers and those with more access to the project can access it. For more details, see <i>Accessing Web User Interfaces from Cloudera Machine Learning</i>.</p> <p>Use 127.0.0.1 as the IP.</p>
CDSW_DOMAIN	The domain on which Cloudera Machine Learning is being served. This can be useful for iframing services, as demonstrated in <i>Accessing Web User Interfaces from Cloudera Machine Learning</i> .
CDSW_CPU_MILLICORES	The number of CPU cores allocated to this engine, expressed in thousandths of a core.
CDSW_MEMORY_MB	The number of megabytes of memory allocated to this engine.
CDSW_IP_ADDRESS	Other engines in the Cloudera Machine Learning cluster can contact this engine on this IP address.

Accessing Environmental Variables from Projects

This topic shows you how to access environmental variables from your code.

Environmental variables are injected into every engine launched for a project, contingent on the scope at which the variable was set (global, project, etc.). The following code samples show how to access a sample environment variable called `DATABASE_PASSWORD` from your project code.

Python

```
import os
database_password = os.environ["DATABASE_PASSWORD"]
```

Appending Values to Environment Variables:

You can also set environment variables to append to existing values instead of replacing them. For example, when setting the `LD_LIBRARY_PATH` variable, you can set the value to `LD_LIBRARY_PATH:/path/to/set`.

Customized Runtimes

This topic explains how custom Runtimes work and when they should be used.



Note: Cloudera, Inc. (“Cloudera”) makes the Custom Runtime feature available to allow its users to add, run, and manage their own container images for their workloads as ‘Custom Runtimes’. Cloudera’s support covers only the integration points of these images with Cloudera Machine Learning, to the extent adding, running, and managing Custom Runtime images fulfill the documented requirements and/or pre-requisites. Cloudera does not provide any support for any other configurations and/or third party software added or installed to the image by customers.

By default, Cloudera Machine Learning Runtimes are preloaded with a few common packages and libraries for R, Python, and Scala. In addition to these, Cloudera Machine Learning also allows you to install any other packages or libraries that are required by your projects. However, directly installing a package to a project as described above might not always be feasible. For example, packages that require root access to be installed, or that must be installed to a path outside /home/cdsd (outside the project mount), cannot be installed directly from the workbench.

For such circumstances, Cloudera Machine Learning allows you to extend the base Docker image and create a new Docker image with all the libraries and packages you require. Site administrators can then add this new image in the allowlist for use in projects.



Note: You will need to remove any unnecessary Cloudera sources or repositories that are inaccessible because of the paywall.

PBJ Custom Runtimes can be built on top of any Ubuntu base image, and users have to install the kernel themselves. However, non-PBJ Runtime images can only be built on top of Cloudera-released non-PBJ Runtime images, and users cannot change the kernel.

Note that this approach can also be used to accelerate project setup across the deployment. For example, if you want multiple projects on your deployment to have access to some common dependencies (package or software or driver) out of the box, or even if a package just has a complicated setup, it might be easier to simply provide users with a Runtime that has already been customized for their project(s).

Related Resources

- The Cloudera Engineering Blog post on *Customizing Docker Images in Cloudera Machine Learning* describes an end-to-end example on how to build and publish a customized Docker image and use it as an engine in Cloudera Machine Learning.
- For an example of how to extend the base engine image to include Conda, see *Installing Additional Packages*.

Creating Customized ML Runtimes

This section walks you through the steps required to create your own custom ML Runtimes based on one of the Cloudera provided ML Runtime images.



Note: In case of PBJ Runtimes, the Custom runtime does not have to be built on the top of a Cloudera provided ML Runtime image.

Create a Dockerfile for the Custom Runtime Image

This topic shows you how to create a Dockerfile for a custom image.

First, select an appropriate source image for your customization. For a non-PBJ Runtime, you must use a Runtime image released by Cloudera. Image tags can be seen on the Session Start page on the user interface when you select a Runtime. The second step when building a customized image is to create a Dockerfile that specifies which packages you would like to install in addition to the base image.

For example, the following Dockerfile installs the telnet package, the sklearn Python package and upgraded base packages on top of an ML Runtime image released by Cloudera.

```
# Dockerfile
# Specify an ML Runtime base image
FROM docker.repository.cloudera.com/cloudera/cdsd/ml-runtime-jupyterlab-pyth
on3.7-standard:2021.12.1-b17
```

```
# Install telnet in the new image
RUN apt-get update && apt-get install -y --no-install-recommends telnet &&
  apt-get clean && rm -rf /var/lib/apt/lists/*
# Upgrade packages in the base image
RUN apt-get update && apt-get upgrade -y && apt-get clean && rm -rf /var/lib
  /apt/lists/*
# Install the python package sklearn
RUN pip install --no-cache-dir sklearn
# Override Runtime label and environment variables metadata
ENV ML_RUNTIME_EDITION="Telnet Edition" \
    ML_RUNTIME_SHORT_VERSION="1.0" \
    ML_RUNTIME_MAINTENANCE_VERSION=1 \
    ML_RUNTIME_DESCRIPTION="This runtime includes telnet and sklearn
and upgraded packages"
ENV ML_RUNTIME_FULL_VERSION="${ML_RUNTIME_SHORT_VERSION}.${ML_RUNTIME_MAI
NTENANCE_VERSION}"
LABEL com.cloudera.ml.runtime.edition=$ML_RUNTIME_EDITION \
    com.cloudera.ml.runtime.full-version=$ML_RUNTIME_FULL_VERSION \
    com.cloudera.ml.runtime.short-version=$ML_RUNTIME_SHORT_VERSION \
    com.cloudera.ml.runtime.maintenance-version=$ML_RUNTIME_MAINTENANCE
_VERSION \
    com.cloudera.ml.runtime.description=$ML_RUNTIME_DESCRIPTION
```

Metadata for Custom ML Runtimes

This topic addresses the metadata for custom Runtimes.

All new custom Runtimes must override the Edition metadata of existing Runtimes. The rest of the metadata can be overridden to communicate the expectations for the consumers of the image. Both the Docker label and the environment variable match in a custom Runtime image. In order to add or register a custom Runtime to a deployment, the user facing metadata combination should be unique in that deployment. For example, of the following, Editor, Edition, Kernel, Version, and Maintenance Version, at least the later should be incremented for adding a next iteration of the same image.

See the following reference table for more details on ML Runtime metadata.

Environment variable	Docker Label	Description	Override in custom non-PBJ runtime	Value in PBJ Runtimes
ML_RUNTIME_METADATA_VERSION	com.cloudera.ml.runtime.runtime-metadata-version	Metadata version	Not allowed	Must be set to 2
ML_RUNTIME_EDITOR	com.cloudera.ml.runtime.editor	CDSW/CML Editor installed in the image.	Allowed	Required
ML_RUNTIME_EDITION	com.cloudera.ml.runtime.edition	Edition of the image, a notion of the Runtime capabilities.	Required	Required
ML_RUNTIME_DESCRIPTION	com.cloudera.ml.runtime.description	Longer description of the Runtime image capabilities.	Recommended	Required
ML_RUNTIME_KERNEL	com.cloudera.ml.runtime.kernel	Main kernel included in the image, e.g., Python 3.8	Not allowed	Required
ML_RUNTIME_SHORT_VERSION	com.cloudera.ml.runtime.short-version	Main version of the image, e.g., 1.0. This shows up as Version in the selection screen.	Recommended	Required
ML_RUNTIME_FULL_VERSION	com.cloudera.ml.runtime.full-version	Full version consists of the short version + maintenance version, e.g., 1.0.1	Optional	Optional

ML_RUNTIME_MAINTENANCE_VERSION	com.cloudera.ml.runtime.maintenance-version	Maintenance version must be an integer, e.g., 1. Only the largest maintenance version of the set of the same short version images are visible for users to select. Increment this number if you create a drop in replacement of an existing Runtime. Start with 1 with a new version or edition.	Recommended	Required
ML_RUNTIME_CUDA_VERSION	com.cloudera.ml.runtime.cuda-version	CUDA version installed in the image.	Not allowed	Do not set
ML_RUNTIME_GIT_HASH	com.cloudera.ml.runtime.git-hash	Git hash of runtime source	Not allowed	Do not set
ML_RUNTIME_GBN	com.cloudera.ml.runtime.gbn	Cloudera internal build number	Not allowed	Do not set

Editor Customization

This topic addresses customizing a third-party editor to work with ML Runtimes.

For a third-party editor to work with ML Runtimes, you must provide a starting script. For example:

```
#!/bin/bash
"$ZEPPELIN_HOME/bin/zeppelin.sh"
```

For Cloudera Machine Learning to interpret this as an editor startup script, you must create a symlink to the editor as `usr/local/bin/ml-runtime-editor`. This will be created in the Dockerfile of the customized runtime.



Note: Third-party editors provide a way to run arbitrary code that is not distributed by Cloudera. Use third-party editors at your own risk. You should absolutely trust the code that you want to run.

Build the New Docker Image

This topic shows you how to use Docker to build a custom image.

A new custom Docker image can be built on any host where Docker binaries are installed, source images are available, and OS repositories and other package repositories are available. To install these binaries, run the following command on the host where you want to build the new image:

```
docker build -t <image-name>:<tag> . -f Dockerfile
```

If your Dockerfile makes any outside connection (for example, `apt-get`, `update`, `pip install`, `curl`), you must add the `--network=host` option to the build command:

```
docker build --network=host -t <image-name>:<tag> . -f Dockerfile
```

Distribute the Image

This topic explains the different methods that can be used to distribute a custom ML Runtime to all the hosts.

Once you have built a new custom ML Runtime, use one of the following methods to distribute the new image to all your Cloudera Machine Learning hosts:

Push the image to a public registry such as DockerHub

For instructions, refer to the Docker documentation *docker push* and *Push images to Docker Cloud*.

Push the image to your company's Docker registry

When using this method, make sure to tag your image with the following schema:

```
docker tag <image-name> <company-registry>/<user-name>/<image-name>:<tag>
```

Once the image has been tagged properly, use the following command to push the image:

```
docker push <company-registry>/<user-name>/<image-name>:<tag>
```

Add the new ML Runtime

Cloudera Machine Learning enables you to add customized ML Runtimes from the Runtime Catalog window.

About this task



Note: You must have system administrator permission to add a new ML Runtime.



Note: If you add a Custom Runtime from a private docker registry, you need to add the docker credentials first to CML. See *Add Docker registry credentials and certificates* for more information.

Procedure

1. Click Runtime Catalog from the Navigation panel.
2. Click the Add Runtime button in the upper right corner.
3. In the Add Runtime window, enter the url of the Runtime Docker image you want to upload.

As ML Runtimes are identified based on certain attributes, metadata (such as Editor, Kernel, Edition, Version, and Maintenance Version) must be unique to add new Customized Runtimes to a deployment. Customized ML Runtimes must have different Edition text compared to Cloudera supported versions.

4. Click Validate.

CML will use the provided URL to fetch the Docker image and validate if it can be used as a customized Runtime.

If the Docker image is successfully validated, CML will display the metadata information of the image. The new customized Runtime will be visible in the Runtime Catalog and accessible over the different workloads.

Related Information

[Add Docker registry credentials and certificates](#)

Limitations

This topic lists some limitations associated with customized ML Runtime images.

- The contents of certain pre-existing standard directories such as /home/cdsw, /tmp, and so on, cannot be modified while creating customized non-PBJ ML Runtimes. This means any files saved in these directories will not be accessible from sessions that are running on customized ML Runtimes.

Workaround: Create a new custom directory in the Dockerfile used to create the customized ML Runtime, and save your files to that directory.

For PBJ Runtimes, note the following limitations:

- PBJ Runtimes work as models only with R and Python kernels.

Add Docker registry credentials and certificates

To enable Cloudera Machine Learning to fetch custom ML Runtimes from a secure repository, as Administrator you need to add Docker registry credentials.

Add Docker Registry Credentials to CML

Create a `kubectl` secret named `regcred` for your secured Docker registry. The following command creates the secret in your Kubernetes cluster:

```
kubectl create secret docker-registry regcred
--docker-server=<server host>
--docker-username=<username>
--docker-password=<password>
-n <compute namespace eg. mlx>
```

The next time the ML Runtime image is pulled, the new secret will be picked up.



Note: Consult the documentation of your chosen docker registry to understand what docker login credentials it expects. The expected credentials might be different from your own username and password. Many registries require docker authentication using tokens or API keys.

```
kubectl create secret docker-registry regcred
--docker-server=<server host>
--docker-username=AWS
--docker-password=$(aws ecr get-login-password --region <region>)
-n <compute namespace eg. mlx>
```



Note: You need to add these credentials to CML again if you back up and then restore your CML workspace.

Pre-Installed Packages in ML Runtimes

Cloudera Machine Learning ships with several base engine images that include Python kernels, and frequently used libraries.



Note: New ML Runtime releases are automatically added to the deployment, if internet connection is available.

As the software and versions listed in *ML Runtimes Pre-installed Packages* are installed in the images, you shall not upgrade these versions as this may lead to dependency conflicts. If any related software in this list needs to be at a higher version, you must use a more recent version of the legacy engine or ML Runtime.

Related Information

[ML Runtimes Pre-installed Packages](#)