

< Previous

SEARCH ALL BLOGS



Next blog post >

TECHNICAL

# Managing Python dependencies for Spark workloads in Cloudera Data Engineering

SHARE

Apache Spark

CDP Private Cloud

CDP Public Cloud

Cloudera Data Platform (CDP)

Data Engineering



by Vijay Karthikeyan

Posted in TECHNICAL | April 30, 2021 7 min read

Editor's  
Choice

**Update August 2021:** Starting with CDE v1.9, you can now use the `python-env` resource (Option 2) for all Python packages, including those dependent on C base libraries such as Pandas, Pyarrow, etc. Use `custom-runtime-image` (Option 3) only for custom libraries & more advanced scenarios.

Apache Spark is now widely used in many enterprises for building high-performance ETL and Machine Learning pipelines. If the users are already familiar with Python then PySpark provides a python API for using Apache Spark. When users work with PySpark they often use existing python and/or custom Python packages in their program to extend and complement Apache Spark's functionality. Apache Spark provides several options to manage these dependencies. For legacy Cloudera CDH, Hortonworks Data Platform (HDP) customers and customers currently using Apache Spark on Cloudera Data Platform (CDP) Private Cloud Base, the **standard way** to manage dependencies are as follows –

- Use the `-py-files` option or `spark.submit.pyFiles` configuration to include Python dependencies as part of the `spark-submit` command.
- For dependencies that need to be made available at runtime, the options are to
  - Install Python dependencies on all nodes in the Cluster
  - Install Python dependencies on a shared NFS mount and make it available on all node manager hosts

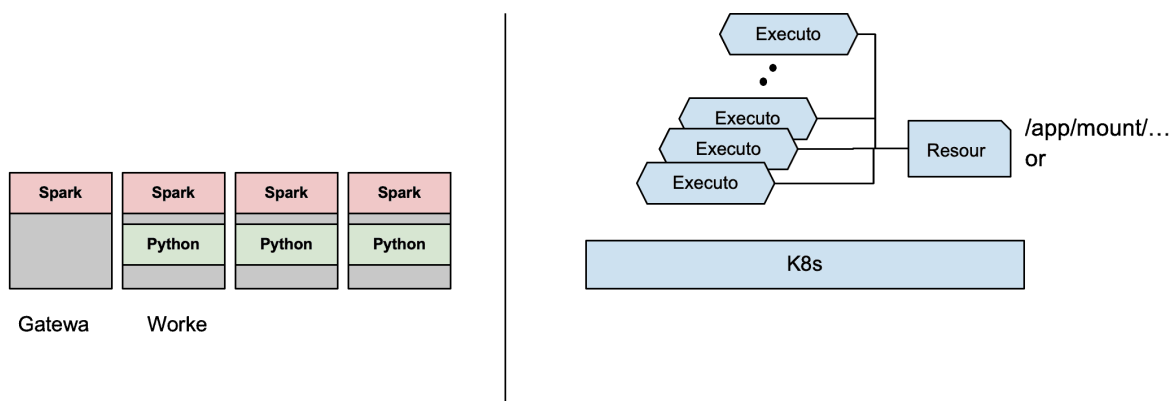


**BUSINESS**  
Generative  
AI for the  
Enterprise



**TECHNICAL**  
Building  
Trust in  
Public  
Sector AI  
Starts with  
Trusting  
Your Data

Package the dependencies using Python Virtual environment or Conda package and ship it with *spark-submit* command using *-archives* option or the *spark.yarn.dist.archives* configuration.



Traditional Hadoop Cluster on

Cloudera Data Engineering

Cloudera Data Engineering (CDE) is a cloud-native service purpose-built for enterprise data engineering teams. CDE is already available in CDP Public Cloud (AWS & Azure) and will soon be available in CDP Private Cloud Experiences. CDE runs Apache Spark on K8S using Apache YuniKorn scheduler. To find out more about CDE review [this](#) article. This blog reviews the available options for managing Python dependencies in Cloudera Data Engineering (CDE) using resources, a [new API abstraction](#) available out of the box to make managing spark jobs and their associated artifacts much easier. Using CDE's APIs allows for easy automation of ETL workload and integration with any CI/CD workflows. We will first start out showing how to run a simple PySpark job in CDE then provide a few options of managing dependencies to help highlight the flexibility of the jobs APIs.

## Getting started with CDE

This is a simple scenario where the Spark job does not require any additional dependencies. All required dependencies (such as the Hive Warehouse connector) are already included in the base image run by CDE and are made available to the Spark program at runtime. In this case, no additional steps are required.

We can use CDE's "spark-submit" to easily submit our PySpark job from a local machine without having to worry about uploading any files to nodes running on the cluster. CDE's APIs automatically generate the required resource behind the scenes, and mount the files to all the Spark pods at /app/mount path.

[Here](#) is an example showing a simple PySpark program querying an ACID table. Create and trigger the job using the [CDE CLI](#).

```
# Using Spark Submit to submit an Ad-Hoc job
cde spark submit pyspark-example-1.py \
```

```
--file read-acid-table.sql
```

Here is a view of the job configuration from the CDE UI showing the *.sql*/file being uploaded under the other dependencies section.

## Jobs / pyspark-example-1

Status	Runs
Ad-Hoc	1

Run History

Configuration

Schedule

Name \*

pyspark-example-1

Application File \*

pyspark-example-1.py

Python Version

☒ Python 3 ☐ Python 2

Other Dependencies

1. read-acid-table.sql

Executors

1

400

1

Application File

Other dependencies

Status: **Succeeded** Job: **pyspark-example-1** Lineage: **Atlas** Duration: 1.1 min Start Time: Apr 1, 2021, 8:02:27 AM

Trends Configuration **Logs** Analysis Spark UI

Select log type: Submitter

stderr stdout Kubernetes **Jobs API**

```

2021-04-01 13:02:27 INFO Run created
2021-04-01 13:02:27 DEBUG Last used time for job 'pyspark-example-1' updated
2021-04-01 13:02:27 DEBUG Last used time for resource 'AutoResource-1617282126546' updated
2021-04-01 13:02:27 DEBUG Workspace copy for mount AutoResource-1617282126546 took 46.542192ms
2021-04-01 13:02:27 DEBUG Built job run workspace: /app/dex/storage/run/220/workspace
2021-04-01 13:02:27 DEBUG TGT generation successful, response: &{Principal:vijay.karthikeyan@cluster-992ts1f.dex.cloudera.site StartUID:0a13b4ef-ade7-4f7b-b35e-1051aa022db SecretName:tgt-secret-vijay.karthikeyan}
2021-04-01 13:02:28 DEBUG Run context created: &{Job:0xc000f48900 WorkDir:/app/dex/storage/run/220/workspace Overrides:cnll Variables:map[] TgtSecretName:tgt-secret-vijay.karthikeyan ResourceWlper:0xc00196a880}
2021-04-01 13:02:28 DEBUG No overrides provided
2021-04-01 13:02:28 DEBUG No variables to interpolate
2021-04-01 13:02:28 DEBUG job-pyspark-example-1, driverTemplate=/app/dex/storage/run/220/workspace/dex-spark-driver-template-5hh8b2wj.yaml
2021-04-01 13:02:28 DEBUG job-pyspark-example-1, executorTemplate=/app/dex/storage/run/220/workspace/dex-spark-executor-template-5hh8b2wj.yaml
2021-04-01 13:02:28 DEBUG Will not detect main class: application file does not have jar file extension
2021-04-01 13:02:28 DEBUG Livy batch create request: {
  "name": "pyspark-example-1-220",
  "file": "local:///app/mount/pyspark-example-1.py",
  "jars": [
    "local:///opt/spark/optional-lib/hive-warehouse-connector-assembly.jar"
  ],
  "pyfiles": [
    "local:///opt/spark/optional-lib/pyspark_hwc.zip"
  ],
  "files": [
    "local:///app/mount/read-acid-table.sql"
  ]
}

```

Annotations in the log:

- Job file uploaded to /app/mount (points to "file")
- Other required dependencies uploaded automatically (points to "pyfiles")
- Dependencies uploaded to /app/mount (points to "files")

## Option 1: Jobs using user-defined Python functions

In some scenarios, the Spark jobs are dependent on homegrown Python packages. This is usually done for easy maintenance and reusability. These dependencies are supplied to the job as .py files or in a packaged format such as Egg or Zip files. For such cases, you have a couple of options for including them in the job submission. These scenarios are illustrated with the **example** below. In this example, the PySpark job has three dependencies (1) a .py file (2) A zip file, and (3) An Egg file which defines functions used by the main application file.

### Option 1a: Include the dependencies in every job

The first option is to include all the files required as part of the job definition. To run this example through CDE CLI, run the following command to trigger the job.

```

# Create CDE job or use cde spark submit
cde spark submit pyspark-example-2a.py \


--py-file file_printDF.py \

--py-file egg-zip/ReadCsvEggFile-1.0-py3.7.egg \

--py-file egg-zip/printPath.zip

```

The job definition from the UI shows that the dependent .py, .zip, and .egg files are uploaded into the "Python, Egg, Zip files" section and are uploaded to the "/app/mount" path inside the container.

Status	Job	Lineage
 Succeeded	<a href="#">pyspark-example-2a</a>	<a href="#">Atlas</a>
Trends	<a href="#">Configuration</a>	Logs
Analysis		

Name \*

Application File \*

Application file  
uploaded here

Python Version


☒ Python 3 ☐ Python 2

Python dependencies  
uploaded here

Python, Egg, Zip files

1. file\_printDF.py
2. ReadCsvEggFile-1.0-py3.7.egg
3. printPath.zip

Job Runs / 242

Status	Job	Lineage	Duration	Start Time
 Succeeded	<a href="#">pyspark-example-2a</a>	<a href="#">Atlas</a>	3.6 min	Apr 1, 2021, 11:30:57 AM
Trends	Configuration	Logs	Analysis	Spark UI

Select log type

Driver

Spark eventsstderr + stdoutPod eventsstdoutstderrTGT loader

```
driver --properties-file /opt/spark/conf/spark.properties --class org.apache.spark.deploy.PythonRunner /app/mount/pyspark-example-2a.py
All files under /app/mount:
['/app/mount/pyspark-example-2a.py', '/app/mount/ReadCsvEggFile-1.0-py3.7.egg', '/app/mount/dex-spark-driver-template-49z7l7h6.yaml', '/app/mount/dex-spark-executor-template-49z7l7h6.yaml', '/app/mount/file_printDF.py',
'/app/mount/printPath.zip']
Print inside fn_printDF
+-----+
| 1| 2| 3|
+-----+
| 4| 5| 6|
| 7| 8| 9|
+-----+

Print inside fn_print_path
Current working directory: /opt/spark/work-dir
Starting sampling
Stopping sampling
```

All files inside /app/mount

**Option 1b: Create a resource & attach it to the jobs  
(recommended)**

If you have a common setup of dependencies, then you can create a resource to upload all the files and mount them to the container at runtime. This way you can manage the **dependencies centrally and re-use the same resource across multiple jobs**.

The below example creates two resources – the first one containing the application file and the second one containing the three dependencies used by the application file.

*# Create a resource for the application file*

```
cde resource create --type files --name pyspark-example-2b-res
cde resource upload --name pyspark-example-2b-resource \
--local-path pyspark-example-2b.py
```

*# Create a resource for the common dependencies*

```
cde resource create --type files --name common-py-files
cde resource upload --name common-py-files \
--local-path file_printDF.py \
--local-path egg-zip/ReadCsvEggFile-1.0-py3.7.egg \
--local-path egg-zip/printPath.zip
```

Here is a view of the uploaded files in the resource:

The screenshot shows the Cloudera Data Engineering interface. On the left is a sidebar with navigation links: 'Job Runs', 'Jobs', and 'Resources'. The main area displays two resource views. The first view is for 'Resources / common-py-files / Files'. It shows a search bar and a table with three files: 'printPath.zip', 'file\_printDF.py', and 'ReadCsvEggFile-1.0-py3.7.egg', all created on April 1, 2021. The second view is for 'Resources / pyspark-example-2b-reso... / Files'. It shows a search bar and a table with one file: 'pyspark-example-2b.py', created on April 1, 2021. Arrows point from the resource names in the breadcrumb to their respective tables.

Name	Created On
printPath.zip	Apr 1, 2021, 11:49:57 AM
file_printDF.py	Apr 1, 2021, 11:49:56 AM
ReadCsvEggFile-1.0-py3.7.egg	Apr 1, 2021, 11:49:56 AM

Name	Created On
pyspark-example-2b.py	Apr 1, 2021, 11:42:34 AM

Run the following command to create the job and trigger it. Note the syntax "*--mount-N-resource*" pointing to the resources created in the earlier step. Also, note that each mounted resource can also be mounted to a specific path. Resources are mounted to the "*/app/mount*" directory which is the case for *pyspark-example-2b-resource*. For the *common-py-files*

resource, the mount prefix is set to "*commonPyFiles/*" which means all the files for this resource will be made available under "*/app/mount/commonPyFiles/*".

### # Create the CDE job

```
cde job create --type spark \  
--application-file pyspark-example-2b.py \  
--mount-1-resource pyspark-example-2b-resource \  
--mount-2-prefix "commonPyFiles/" \  
--mount-2-resource common-py-files \  
--name pyspark-example-2b
```

### # Run the job

```
cde job run --name pyspark-example-2b
```

Here is a view from the logs showing that the resource is mounted as */commonPyFiles* and made available during job execution.

Job Runs / 246

The screenshot shows the job details for 'pyspark-example-2b' with a status of 'Succeeded'. The 'Logs' tab is selected, showing the 'Driver' log type. The log content includes the Spark command and the output of the application. Annotations highlight the mounting of resources:

- A green arrow points from the log line `'/app/mount/commonPyFiles/ReadCsvEggFile-1.0-py3.7.egg',` to the label **mount-1-resource @/app/mount**.
- A blue arrow points from the log line `'/app/mount/commonPyFiles/printPath.zip'` to the label **mount-2-resource @/app/mount/<mount-2-prefix>**.

The log output shows a directory listing:

```
Print inside fn_printDF  
-----  
| 1| 2| 3|  
-----  
| 4| 5| 6|  
| 7| 8| 9|  
-----  
Print inside fn_print_Path  
Current working directory: /opt/spark/work-dir  
Starting sampling  
Stopping sampling
```

## Option 2: Jobs using pure Python libraries

When a job is dependent on pure Python packages users can create a resource of type "*python-env*" with the *requirements.txt* file. This file contains the list of items to be installed using *pip* install. When a resource of type "*python-env*", CDE internally creates a virtual environment using the provided requirements.txt file and mounts it with the job. You can now launch jobs with this resource as the *python-env-resource-name*. Let us review an example to understand this better.

In this [example](#), the python program uses the *boto3* Python API to retrieve the database password from AWS Secrets Manager before kicking off the Spark program. The default CDE image does not include this package hence we will first create a resource of type "*python-env*" and upload the *requirements.txt* file to it.

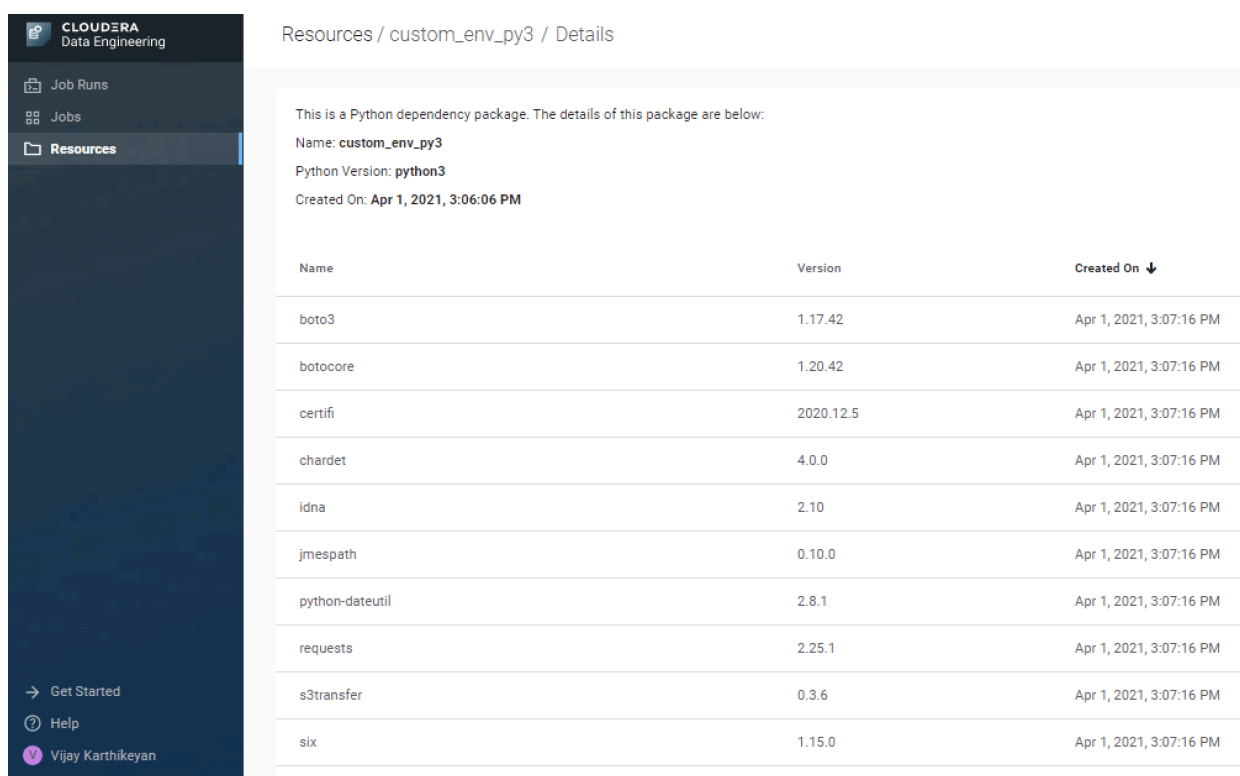
```
# Create resource of type python-env
```

```
cde resource create --name custom_env_py3 --type python-env
```

```
# Upload the requirements.txt file to the resource
```

```
cde resource upload --name custom_env_py3 \  
--local-path requirements.txt
```

Review the status of the resource from UI or use the command *cde resource describe* from CDE CLI to get the status. If you try to create a job immediately then you may get the error "*Error: create job failed: can not use resource 'custom\_env\_py3' in status 'building'*". Here is a view of the resource from the UI.



The screenshot shows the Cloudera Data Engineering (CDE) UI. On the left is a sidebar with navigation links: Job Runs, Jobs, Resources (selected), Get Started, Help, and a user profile for Vijay Karthikeyan. The main panel is titled 'Resources / custom\_env\_py3 / Details'. It contains a description: 'This is a Python dependency package. The details of this package are below:'. Below this, it lists the resource name as 'custom\_env\_py3', the Python version as 'python3', and the creation time as 'Apr 1, 2021, 3:06:06 PM'. A table follows, listing the installed Python packages with their names, versions, and creation times.

Name	Version	Created On ↓
boto3	1.17.42	Apr 1, 2021, 3:07:16 PM
botocore	1.20.42	Apr 1, 2021, 3:07:16 PM
certifi	2020.12.5	Apr 1, 2021, 3:07:16 PM
chardet	4.0.0	Apr 1, 2021, 3:07:16 PM
idna	2.10	Apr 1, 2021, 3:07:16 PM
jmespath	0.10.0	Apr 1, 2021, 3:07:16 PM
python-dateutil	2.8.1	Apr 1, 2021, 3:07:16 PM
requests	2.25.1	Apr 1, 2021, 3:07:16 PM
s3transfer	0.3.6	Apr 1, 2021, 3:07:16 PM
six	1.15.0	Apr 1, 2021, 3:07:16 PM

Create and trigger the job. You should now see the custom resource is mounted inside the container

```
# Create job
```

```
cde job create --type spark \  
--local-path requirements.txt
```



```

--application-file pyspark-example-3.py \
--python-env-resource-name custom_env_py3 \
--name pyspark-example-3 \
--log-level INFO

# Run the job

cde job run --name pyspark-example-3

```

The screenshot shows the Cloudera Data Engineering (CDE) Job Runs interface. On the left is a sidebar with 'Job Runs', 'Jobs', and 'Resources'. The main panel displays a job run for 'pyspark-example-3' with a status of 'Succeeded', duration of '3.3 min', and start time of 'Apr 1, 2021, 8:06:45 PM'. Below this, there are tabs for 'Trends', 'Configuration', 'Logs', 'Analysis', and 'Spark UI'. The 'Logs' tab is selected, showing a log stream with various DEBUG and INFO messages. A 'Select log type' dropdown is set to 'Submitter'. On the right, there's a 'Job Runs / 270' summary card with a 'Logs' tab. Below it, a 'Select log type' dropdown is set to 'Driver'. A red arrow points from the text 'Output from boto3' to the log output, and another red arrow points from the text 'Custom resource gets mounted here' to a specific log entry.

## Option 3: Jobs using custom libraries and packages

If the job accesses more custom libraries that require RPM packages or other compiled C libraries then you can build a custom docker image built on top of the CDE base image to run the jobs. Some libraries such as pandas, Pyarrow which are frequently used with PySpark are good examples of this scenario (in the future all python libraries would be handled through *venv* mentioned in Option 2, but for now we will use this as an example for option 3).

In **this** example, CDE is used to execute a Machine learning scoring job that is dependent on packages such as pandas, NumPy, XGBoost, and more using a custom container. To deploy this example, follow these steps –

1. Get the base image name & tag from the Cloudera docker repository

```

# Login to the Cloudera Docker Repo
docker login https://container.repository.cloudera.com \
-u $CLDR_REPO_USER -p $CLDR_REPO_PASS

# Check the Available catalogs

curl -u $CLDR_REPO_USER:$CLDR_REPO_PASS \

```

```
-X GET https://container.repository.cloudera.com/v2/_catalog

# Check the images list & select an image

curl -u $CLDR_REPO_USER:$CLDR_REPO_PASS \

-X GET https://container.repository.cloudera.com/v2/cloudera/d
```

2. Build the custom container and publish it to a container registry. Dockerfile used in this example can be found [here](#).

```
# Build Container
docker build --network=host -t vka3/cde:cde-runtime-ml . -f Do

# Push to Container registry

docker push docker.io/vka3/cde:cde-runtime-ml
```

3. On the CDE cluster, create a resource of type "custom-runtime-image". If you need to use credentials for the docker repository then review these additional [instructions](#) from the Cloudera documentation to follow additional steps.

```
# Create the resource for the Docker container

cde resource create --type="custom-runtime-image" \
  --image-engine="spark2" \
  --name="cde-runtime-ml" \
  --image="docker.io/vka3/cde:cde-runtime-ml"
```

4. Create a job using the newly created resource.

```
# Create the job

cde job create --type spark --name ml-scoring-job \
  --runtime-image-resource-name cde-runtime-ml \
  --application-file ./ml-scoring.py
```

5. When the job is run, the job will run using the custom container image.

```
# Run job

cde job run --name ml-scoring-job
```

**CLOUDERA**  
Data Engineering

**Job Runs**

Jobs

Resources

Job Runs / 211

```
{
  "CreationTimestamp": "2021-03-29T13:31:53Z",
  "Message": "Started container event-log-reader"
},
{
  "CreationTimestamp": "2021-03-29T13:31:53Z",
  "Message": "Pulling image \"docker.io/vka3/cde:cde-runtime-m1\""
},
{
  "CreationTimestamp": "2021-03-29T13:32:33Z",
  "Message": "Successfully pulled image \"docker.io/vka3/cde:cde-runtime-m1\""
},
{
  "CreationTimestamp": "2021-03-29T13:32:49Z",
  "Message": "Created container spark-container"
},
}
```

This post reviewed the available options in CDE to manage Python dependencies for your PySpark jobs. CDE provides flexible options for fully operationalizing your data engineering pipelines and is fully integrated with Shared Data Experience for comprehensive security and governance. [Try out Cloudera Data Engineering today!](#)

The following folks all contributed to the blog through reviews, edits and suggestions: Shaun Ahmadian, Jeremy Beard, Ian Buss

## References:

- Cloudera Data Engineering (CDE) documentation – <https://docs.cloudera.com/data-engineering/cloud/index.html>
- Cloudera Community Articles on CDE – [https://community.cloudera.com/t5/Community-Articles/tkb-p/CommunityArticles/label-name/cloudera%20data%20engineering%20\(cde\)](https://community.cloudera.com/t5/Community-Articles/tkb-p/CommunityArticles/label-name/cloudera%20data%20engineering%20(cde))
- Sample Code used in the post – <https://github.com/karthikeyanvijay/cdp-cde>
- Setup CDE CLI with Git Bash – <https://community.cloudera.com/t5/Community-Articles/Setup-CDE-CLI-with-Git-Bash-on-Windows/ta-p/312808>
- CDE custom runtime image build with AWS ECR & CodeBuild – <https://community.cloudera.com/t5/Community-Articles/CDE-custom-runtime-image-build-with-AWS-ECR-amp-CodeBuild/ta-p/314762>



**Vijay Karthikeyan**  
Senior Solutions Architect  
[More by this author >](#)

## Leave a comment

Your email address will not be published. Links are not permitted in comments.

☐ Save my name, and email in this browser for the next time I comment.

Please leave a comment here...

POST COMMENT



inAbout

Products

Solutions

Services & Support

## Contact Us

US: +1 888 789 1488

Outside the US: +1 650 362 0488

© 2024 Cloudera, Inc. All rights reserved.

[Terms & Conditions](#) | [Privacy Policy and Data Policy](#) | [Cookie Preferences](#)

**Apache Hadoop** and associated open source project names are trademarks of the **Apache Software Foundation**. For a complete list of trademarks, [click here](#).