Cloudera Data Engineering 1.5.5

Data Engineering deployment architecture

Date published: 2020-07-30
Date modified: 2025-11-08

https://docs.cloudera.com/

Legal Notice

Contents

Recommendations for scaling Cloudera Data Engineering deployments

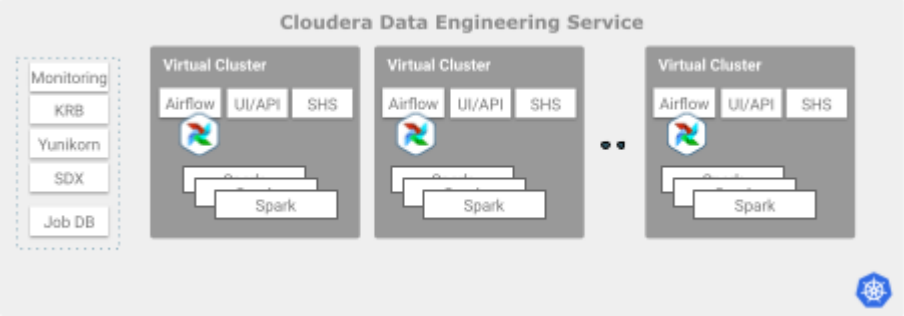
Your business might experience a sudden increase or drop in demand, due to which your Cloudera Data Engineering deployment needs to autoscale. You can scale your Cloudera Data Engineering deployment by either adding new instances of a Cloudera Data Engineering Service or Virtual Cluster, or by adding additional resources to the existing ones.

You can scale your Cloudera Data Engineering deployment the following ways:

- Vertically – More resources are provisioned within the same instance of a Cloudera Data Engineering Service or Virtual Cluster.
- Horizontally – New instances of Cloudera Data Engineering Service or Virtual Cluster are provisioned.

The following image describes the key components of a typical Cloudera Data Engineering service deployment

Figure 1: Cloudera Data Engineering Service deployment components



Virtual Clusters provide isolated, auto-scaling compute capacity to run Spark and Airflow jobs. You can use Virtual Clusters to isolate individual teams or lines of business by using user-based Access Control Lists (ACLs).

Cluster-wide pod limits

Every Cloudera Data Engineering on premises deployment operates on a Kubernetes cluster that has a global capacity. The total number of pods, including Spark drivers, executors, and all platform services, is limited by the platform Kubernetes Admission Controller. This limit is shared by all the services running on the platform, not just Cloudera Data Engineering. When you reach this limit, the Admission Controller rejects new pod creation requests, and the Jobs fail to start. The following error is displayed in the logs:

```
failed calling webhook admission-controller.k8tz.io
```

For more accurate assessment and guidance based on your system specification, reach out to Cloudera Support.

Spark workload scaling

Vertical scaling for a single job

Vertical scaling involves allocating a large number of resources, specifically executors, to a single large Spark job. By default, Cloudera Data Engineering Spark jobs are enabled with dynamic resource allocation. You can specify the following parameters that define the boundaries of scaling:

- Initial executor count
- Minimum executor count
- Maximum executor count

If all these values are set to the same value, the Spark job will run with static allocation. For more information, see the Apache Spark documentation - Dynamic Allocation.

The actual scaling achieved by a single job depends on the complexity of its application logic and the dynamics of the system workload. For more accurate assessment and guidance based on your system specification, reach out to Cloudera support.

Horizontal scaling for concurrent jobs

Horizontal scaling involves running a large number of smaller Spark jobs simultaneously. This scaling pattern is not limited by cluster resources but by the API Server. The default and recommended simultaneous job submission limit for Spark jobs is 60 per Virtual Cluster. Concurrency of actively running jobs can also exceed 60.

Important: This limit of 60 jobs per Virtual Cluster is applicable only for Spark jobs. This is not applicable for Airflow jobs.

Distribute simultaneous submission of jobs over time or horizontally scale across multiple Virtual Clusters. Cloudera does not recommend increasing the submission limit, for example, to 120. This action can cause the API server to become unstable and crash, often due to OutOfMemory errors, resulting in the following issues:

- Failure of all the Jobs that were in the process of being submitted.
- Orphaned Spark driver pods stuck in the initializing state indefinitely, requiring manual administrative cleanup.
- Significant delays in the Cloudera Data Engineering UI and API caused by high concurrency even before a crash. jobs appearing to skip the **Running** state, and the Spark History Server becoming unavailable.

Logging and shuffle data

For additional information on logging and shuffle configurations, see General guidelines.

Spark workload stability and longevity

Cloudera Data Engineering on premises does not provide fault tolerance for the Spark driver pod. If the node hosting the driver pod fails or the pod is otherwise restarted, the job fails and will not automatically recover.

Cloudera recommends implementing state management for long duration jobs at the application, such as checkpointing to allow recovery from a specific point in time.

Virtual Cluster scaling

Each Cloudera Data Engineering Virtual Cluster consumes a fixed amount of platform resources for its own infrastructure pods, such as the job submission service, Airflow services, API servers, before any Cloudera Data Engineering jobs are run. A single Virtual Cluster infrastructure makes a baseline resource request of approximately 12 cores and 32 GB of memory. For more information about hardware requirements, see Additional resource requirements for Cloudera Data Engineering for ECS or Cloudera Data Engineering hardware requirements for OCP.

The maximum number of Virtual Clusters is not a fixed number but is constrained by the available hardware. In an on premises environment with limited resources, the fixed cost per Virtual Cluster is a significant factor.

No limit on resource files in job runs

As on premises, no limit exists for referencing a resource file in job runs in Cloudera Data Engineering on premises, you can reference the same resource file multiple times within a single job run.

Related Information

Creating virtual clusters

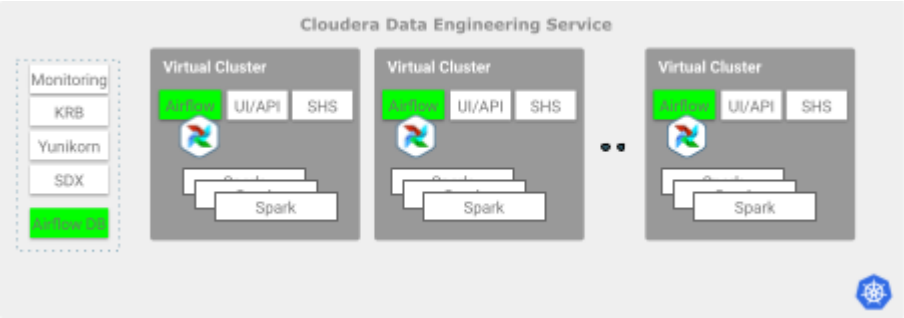
Apache Airflow scaling and tuning considerations

You must consider certain limitations related to the deployment architecture and guidelines for scaling and tuning of the deployment while creating or running Airflow jobs (DAGs).

Cloudera Data Engineering deployment architecture

If you use Airflow to schedule multi-step pipelines, develop multi-step pipelines, or both, you must consider deployment limitations to scale your environment.

Figure 2: Cloudera Data Engineering deployment architecture



Consider the following guidelines to decide when to scale horizontally, given the total number of loaded DAGs, the number of concurrent tasks, or both:

Cloudera Data Engineering service guidelines

The number of Airflow jobs that can run at the same time is limited by the number of parallel tasks that are triggered by the associated DAGs. The number of tasks can reach up to 250-300 when running in parallel per Cloudera Data Engineering service.

Cloudera recommends creating more Cloudera Data Engineering services to increase the possible number of concurrent Airflow tasks beyond the limits.

Virtual Cluster guidelines

Airflow Task Concurrency

The number of Airflow tasks that can physically run at the same time is determined by the hardcoded platform limit and the resources allocated to the Virtual Cluster.

- Each Virtual Cluster has a maximum parallel task execution limit of 250. While thousands of DAGs can be submitted, the Airflow scheduler places them in a queue, and only 250 tasks are allowed to run in parallel across the Virtual Cluster at any given time.
- The number of concurrently running Airflow task pods is limited by the Virtual Cluster aggregate resource quota. Cloudera Data Engineering keeps the tasks in queue until the existing pods finish and free up CPU and memory. This limit is often the effective ceiling, even if the limit is below the 250-task platform limit. The default Airflow worker pod has the following resource request limits:
  - CPU requests = 1
  - CPU limits = No limit
  - Memory requests = 2 Gi
  - Memory limits = 2 Gi

For more information, see Automating data pipelines using Apache Airflow in Cloudera Data Engineering.

Important: Administrators must configure the Virtual Cluster Guaranteed and Maximum CPU and memory quotas to align with their desired Airflow task concurrency, up to the maximum limit.

Strategies for Scaling Airflow

To increase Airflow task concurrency, administrators have the following options:

- **Vertical Scaling (Reaching the Platform Limit)** – This strategy involves increasing the Guaranteed and Maxi mum CPU or memory quotas for a single Virtual Cluster. A Virtual Cluster is often resource-bound. Vertical scaling provides the necessary resources to run more concurrent tasks, allowing you to scale up to the maximum limit.
- **Horizontal Scaling (Beyond the Platform Limit)** – Once a single Virtual Cluster has enough resources to hit the maximum limit, it becomes platform-limited. In this case, scaling the total concurrency further is only possible by distributing Airflow DAGs across multiple, separate Virtual Clusters.

Important: Cloudera recommends creating more Virtual Clusters to load additional Airflow DAGs.

General guidelines

Learn more about general guidelines, known issues, and limitations while scaling Cloudera Data Engineering deployments.

Typically, the total physical resources of a cluster are available for all Spark and Airflow jobs. However, in Cloudera Data Engineering on premises, capacity is consumed in a stack, and Cloudera Data Engineering jobs can only access the remaining unused resources. The following layers of consumption exist:

- **Physical hardware** – The total bare-metal or virtual machine capacity.
- **Platform overhead** – Resources consumed by the host Operating System, Kubernetes services, and platform-level daemonsets for storage and networking.
- **Cloudera Data Services on premises overhead** – Resources consumed by the Cloudera Data Services on premises infrastructure, including the Cloudera Control Plane and Cloudera Shared Data Experience services.
- **Cloudera Data Engineering Service and Virtual Cluster infrastructure overhead** – A fixed amount of resources consumed by the Cloudera Data Engineering service and each Virtual Cluster's internal pods such as Airflow or API server.
- **Available Job Capacity** – The remaining resources available to run Spark executors and Airflow task pods.

Limitations

- **Resource fragmentation** – Jobs can remain in the Pending state even when monitoring dashboards indicate free resources in the cluster. This often occurs when the available capacity is scattered across multiple nodes in small chunks, none of which are large enough to host the job. The following resource fragmentation cases exist:
  - **CPU Fragmentation** – For example, a cluster might have 8 available CPU cores, but if this capacity is distributed as 0.5 cores on 16 different nodes, a Spark job requesting a single 4-core executor cannot be scheduled. The job remains in the Pending state because no single node can satisfy its request.
  - **Memory Fragmentation** – For example, a cluster might have 100 GB of free memory, but if this memory is distributed into 10 GB chunks on 10 different nodes, a job requesting an executor with 20 GB of RAM cannot be scheduled.

Important: When jobs remain in the Pending state due to fragmentation, the most effective solution is to request smaller executors. For example, a job requesting 1-core executors is far more likely to be scheduled, as it can fill the gaps of fractional CPUs left behind by larger jobs. This can lead to higher overall cluster utilization. If you need further assistance to assess the logging requirements for your workloads, reach out to Cloudera support.

- **Control Log Ingestion Rate** – Platform logging services have a maximum ingestion capacity of Disk Write Rate in the order of 450 Kib/second per pod. To ensure log reliability, Spark jobs must not generate logs at an excessive rate. If a job log output significantly exceeds this rate, it can overwhelm the logging pipeline and result in log loss.
- **Memory Overhead (Java or Scala vs Python)** – For memory-critical workloads, the implementation language can matter. The lower the overhead of the job, the more executors can be scheduled. For example, for memory-intensive workloads, the Java or Scala job required a 30% memory overhead for stable execution, while the equivalent Python job required a 40% overhead. The lower overhead of the Java or Scala job allowed more executors to be scheduled, improving concurrency.

  Configuration override – Memory overhead can be configured per job to percentage or absolute value.

  - **Job Level Configuration** – You can set these parameters directly in the Spark configurations for each job.
    - General overhead factor

      ```
      spark.kubernetes.memoryOverheadFactor=0.2
      ```

      This sets a 20% overhead for both driver and executor.
    - Executor-specific overhead

      ```
      spark.executor.memoryOverhead=2g
      ```

      This sets a fixed 2 GB overhead for executors regardless of the executor memory size.
    - Driver-specific overhead

      ```
      spark.driver.memoryOverhead=1g
      ```

      This sets a fixed 1 GB overhead for the driver.
  - **Virtual Cluster Level Configuration** – The memory overhead factors are defined at the virtual cluster level and can be adjusted there.
    - defaultMemoryOverheadFactor (for regular JVM jobs)
    - nonJVMMemoryOverheadFactor (for PySpark jobs)

Configuring Spark jobs for large shuffle data

Learn about configuring the Spark jobs to use the persistent volumes for shuffle data to improve the performance or handle the huge shuffle data.

About this task

By default, Cloudera Data Engineering service does not use any persistent volume for shuffle data causing the shuffle data to spill over to the local disks. If the local disk space is not sufficient or local disk performance decreases, you must configure the Spark jobs to use a persistent volume for shuffle data at the job level.

Note: You can select any storage class for the local volume depending on your requirement and the storage classes available in your Kubernetes cluster.

Before you begin

- You must have a compatible storage class that supports local volumes.

Procedure

1. Copy the following manifest into a new file, named cde-storageclass.yaml:

   ```
   apiVersion: v1
   ```

```
kind: Namespace
metadata:
  name: local-path-storage-cde
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: local-path-provisioner-service-account
  namespace: local-path-storage-cde

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: local-path-provisioner-role-cde
rules:
  - apiGroups: [ "" ]
    resources: [ "nodes", "persistentvolumeclaims", "configmaps" ]
    verbs: [ "get", "list", "watch" ]
  - apiGroups: [ "" ]
    resources: [ "endpoints", "persistentvolumes", "pods" ]
    verbs: [ "*" ]
  - apiGroups: [ "" ]
    resources: [ "events" ]
    verbs: [ "create", "patch" ]
  - apiGroups: [ "storage.k8s.io" ]
    resources: [ "storageclasses" ]
    verbs: [ "get", "list", "watch" ]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: local-path-provisioner-bind-cde
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: local-path-provisioner-role-cde
subjects:
  - kind: ServiceAccount
    name: local-path-provisioner-service-account
    namespace: local-path-storage-cde

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: local-path-provisioner-cde
  namespace: local-path-storage-cde
spec:
  replicas: 1
  selector:
    matchLabels:
      app: local-path-provisioner-cde
  template:
    metadata:
      labels:
        app: local-path-provisioner-cde
    spec:
      serviceAccountName: local-path-provisioner-service-account
      containers:
        - name: local-path-provisioner
          image: <YOUR_REGISTRY>/cloudera_thirdparty/rancher/local-path-
provisioner:v0.0.31
          imagePullPolicy: IfNotPresent
```

```
          command:
            - local-path-provisioner
            - --debug
            - start
            - --config
            - /etc/config/config.json
            - --provisioner-name
            - rancher.io/local-path-cde
          volumeMounts:
            - name: config-volume
              mountPath: /etc/config/
          env:
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
      volumes:
        - name: config-volume
          configMap:
            name: local-path-config

---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-path-cde
provisioner: rancher.io/local-path-cde
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Delete

---
kind: ConfigMap
apiVersion: v1
metadata:
  name: local-path-config
  namespace: local-path-storage-cde
data:
  config.json: |-
    {
            "nodePathMap":[
            {
                    "node":"DEFAULT_PATH_FOR_NON_LISTED_NODES",
                    "paths":["/ecs/ecs/local-storage-cde"]
            }
            ]
    }
  setup: |-
    #!/bin/sh
    while getopts "m:s:p:" opt
    do
      case $opt in
      p)
      absolutePath=$OPTARG
      ;;
      s)
      sizeInBytes=$OPTARG
      ;;
      m)
      volMode=$OPTARG
      ;;
      esac
    done

    mkdir -m 700 -p ${absolutePath}
```

```
    chown 1345:1345 ${absolutePath}
  teardown: |-
    #!/bin/sh
    while getopts "m:s:p:" opt
    do
      case $opt in
      p)
      absolutePath=$OPTARG
      ;;
      s)
      sizeInBytes=$OPTARG
      ;;
      m)
      volMode=$OPTARG
      ;;
      esac
    done

    rm -rf ${absolutePath}
  helperPod.yaml: |-
    apiVersion: v1
    kind: Pod
    metadata:
      name: helper-pod
    spec:
      containers:
      - name: helper-pod
        image: <YOUR_REGISTRY>/cloudera_thirdparty/hardened/busybox:glibc-
1.37.0-r0-202410311742
        imagePullPolicy: IfNotPresent
        securityContext:
          privileged: true
          runAsUser: 0         # Run as root user
          runAsGroup: 0        # Use root group
```

2. In the cde-storageclass.yaml file that you have created, locate and replace the <*YOUR REGISTRY*> placeholder with your container registry path.
3. Open a terminal with kubectl access to your cluster and run the following command to set up the storage class for Cloudera Data Engineering:

```
kubectl apply -f cde-storageclass.yaml
```

4. When creating a new job or editing an existing job in the Cloudera Data Engineering UI, add the required configurations in the Job details Configurations field.

   For example, if you use the local-path-cde local volume storage class and require a volume size of up to 10 Gi, add the following configurations:

Configuration key	Value
spark.kubernetes.executor.volumes.persistentVolumeClaim.spark-local-dir-1.mount.path	/data
spark.kubernetes.executor.volumes.persistentVolumeClaim.spark-local-dir-1.mount.readOnly	false
spark.kubernetes.executor.volumes.persistentVolumeClaim.spark-local-dir-1.options.claimName	OnDemand
spark.kubernetes.executor.volumes.persistentVolumeClaim.spark-local-dir-1.options.sizeLimit	10Gi
spark.kubernetes.executor.volumes.persistentVolumeClaim.spark-local-dir-1.options.storageClass	local-path-cde

Related Information

Creating jobs in Cloudera Data Engineering