Autoscaling clusters

ive: Understand how to configure and use Dataproc autoscaling to automatically and dynamically resize clus
t workload demands.

What is autoscaling?

Estimating the "right" number of cluster workers (nodes) for a workload is difficult, and a single cluster size for an entire pipeline is often not ideal. User-initiated Cluster Scaling (/dataproc/docs/concepts/configuring-clusters/scaling-clusters) partially addresses this challenge, but requires monitoring cluster utilization and manual intervention.

The Dataproc AutoscalingPolicies API (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies) provides a mechanism for automating cluster resource management and enables cluster autoscaling. An `Autoscaling Policy` is a reusable configuration that describes how clusters using the autoscaling policy should scale. It defines scaling boundaries, frequency, and aggressiveness to provide fine-grained control over cluster resources throughout cluster lifetime.

When to use autoscaling

Use autoscaling:

✓ on clusters that store data in external services, such as Cloud Storage (/storage/docs) or BigQuery (/bigquery/docs)

✓ on clusters that process many jobs

✓ to scale up single-job clusters

Autoscaling is **not** recommended with/for:

- **High Availability Clusters**: Instead, use standard clusters, which are more stable after successive resizing operations.

- **HDFS:** Autoscaling is not intended for scaling on-cluster HDFS. If you use autoscaling with HDFS, make sure that the minimum number of primary workers is sufficient to handle all HDFS data. Also realize that decommissioning HDFS Datanodes can delay the removal of workers.
- **YARN Node Labels:** Autoscaling does not support YARN Node Labels (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/NodeLabel.html), nor the property `dataproc:am.primary_only` due to YARN-9088 (https://issues.apache.org/jira/browse/YARN-9088). YARN incorrectly reports cluster metrics when node labels are used.
- **Spark Structured Streaming:** Autoscaling does not support Spark Structured Streaming (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html) (see Autoscaling and Spark Structured Streaming (#autoscaling_and_spark_structured_streaming)).
- **Idle Clusters:** Autoscaling is not recommended for the purpose of scaling a cluster down to minimum size when the cluster is idle. Since creating a new cluster is as fast as resizing one, consider deleting idle clusters and recreating them instead. The following tools support this "ephemeral" model:

  ✓ Use Dataproc Workflows (/dataproc/docs/concepts/workflows/overview) to schedule a set of jobs on a dedicated cluster, and then delete the cluster when the jobs are finished. For more advanced orchestration, use Cloud Composer (/composer/docs), which is based on Apache Airflow (https://airflow.apache.org/).

  ✓ For clusters that process ad-hoc queries or externally scheduled workloads, use Cluster Scheduled Deletion (/dataproc/docs/concepts/configuring-clusters/scheduled-deletion) to delete the cluster after a specified idle period or duration, or at a specific time.

Enabling autoscaling

To enable autoscaling on a cluster:

1. Create an autoscaling policy (#create_an_autoscaling_policy).
2. Either:
   a. Create an autoscaling cluster (#create_an_autoscaling_cluster), or
   b. Enable autoscaling on an existing cluster (#enable_autoscaling_on_an_existing_cluster).

Create an autoscaling policy

gcloud commandREST API (#rest-api)Console (#console)

You can use the `gcloud dataproc autoscaling-policies import` (/sdk/gcloud/reference/dataproc/autoscaling-policies/import) command to create an autoscaling policy. It reads a local YAML (https://yaml.org/) file that defines an autoscaling policy. The format and content of the file should match config objects and fields defined by the autoscalingPolicies (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies) REST API.

The following YAML example defines a policy that specifies all required fields. It also provides `maxInstances` values for both primary and secondary (preemptible) workers, and also specifies a 4-minute `cooldownPeriod` (the default is 2 minutes).

```
workerConfig:
  maxInstances: 100
secondaryWorkerConfig:
  maxInstances: 50
basicAlgorithm:
  cooldownPeriod: 4m
  yarnConfig:
    scaleUpFactor: 0.05
    scaleDownFactor: 1.0
    gracefulDecommissionTimeout: 1h
```

Here is another YAML example that specifies all optional and required autoscaling policy fields.

```
workerConfig:
  minInstances: 2
  maxInstances: 100
  weight: 1
secondaryWorkerConfig:
  minInstances: 0
  maxInstances: 100
  weight: 1
basicAlgorithm:
  cooldownPeriod: 4m
  yarnConfig:
    scaleUpFactor: 0.05
    scaleDownFactor: 1.0
    scaleUpMinWorkerFraction: 0.0
```

```
  scaleDownMinWorkerFraction: 0.0
  gracefulDecommissionTimeout: 1h
```

Run the following `gcloud` command from a local terminal or in Cloud Shell (https://console.cloud.google.com/?cloudshell=true) to create the autoscaling policy. Provide a name for the policy. This name will become the policy `id`, which you can use in later `gcloud` commands to reference the policy. Use the `--source` flag to specify the local file path and file name of the autoscaling policy YAML file to import.

```
$ gcloud dataproc autoscaling-policies import policy-name \
    --source=filepath/filename.yaml \
    --region=region
```

Create an autoscaling cluster

After creating an autoscaling policy (#create_an_autoscaling_policy), create a cluster that will use the autoscaling policy. The cluster must be in the same region (/dataproc/docs/concepts/regional-endpoints) as the autoscaling policy.

gcloud commandREST API (#rest-api)Console (#console)

Run the following `gcloud` command from a local terminal or in Cloud Shell (https://console.cloud.google.com/?cloudshell=true) to create an autoscaling cluster. Provide a name for the cluster, and use the `--autoscaling-policy` flag to specify the ***policy id*** (the name of the policy you specified when you created the policy (#create_an_autoscaling_policy)) or the policy ***resource URI (resource name)*** (/apis/design/resource_names) (see the AutoscalingPolicy `id` and `name` fields (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#resource-autoscalingpolicy)).

```
$ gcloud dataproc clusters create cluster-name \
    --autoscaling-policy=policy id or resource URI \
    --region=region
```

The full resource name (or URI) of a policy can be obtained by running the following `gcloud` command:

```
$ gcloud dataproc autoscaling-policies \
    describe policy-id \
    --region=region
```

Enable autoscaling on an existing cluster

After creating an autoscaling policy (#create_an_autoscaling_policy), you can enable the policy on an existing cluster in the same region (/dataproc/docs/concepts/regional-endpoints).

gcloud commandREST API (#rest-api)Console (#console)

Run the following `gcloud` command from a local terminal or in Cloud Shell (https://console.cloud.google.com/?cloudshell=true) to enable an autoscaling policy on an existing cluster. Provide the cluster name, and use the `--autoscaling-policy` flag to specify the ***policy id*** (the name of the policy you specified when you created the policy (#create_an_autoscaling_policy)) or the policy ***resource URI (resource name)*** (/apis/design/resource_names) (see the AutoscalingPolicy `id` and `name` fields (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#resource-autoscalingpolicy)).

```
$ gcloud dataproc clusters update cluster-name \
    --autoscaling-policy=policy id or resource URI \
    --region=region
```

The full resource name (or URI) of a policy can be obtained by running the following `gcloud` command:

```
$ gcloud dataproc autoscaling-policies \
    describe policy-id \
    --region=region
```

How autoscaling works

Autoscaling checks cluster Hadoop YARN metrics (#hadoop_yarn_metrics) as each "cooldown" period elapses to determine whether to scale the cluster, and, if so, the magnitude of the update.

1. On each evaluation, autoscaling examines pending and available cluster memory averaged over the last `cooldown_period` in order to determine the exact change needed to the number of workers:

   `exact Δworkers = avg(pending memory - available memory) / memory per node manager`

   - `pending memory` is a signal that the cluster has tasks queued but not yet executed and may need to be scaled up to better handle its workload.
   - `available memory` is a signal that the cluster has extra bandwidth on healthy nodes and may need to be scaled down to conserve resources.
   - See Autoscaling with Hadoop and Spark (#autoscaling_with_apache_hadoop_and_apache_spark) for additional information on these Apache Hadoop YARN metrics.

2. Given the exact change needed to the number of workers, autoscaling uses a `scaleUpFactor` or `scaleDownFactor` to calculate the actual change to the number of workers:

   ```
   if exact Δworkers > 0:
     actual Δworkers = ROUND_UP(exact Δworkers * scaleUpFactor)
     # examples:
     # ROUND_UP(exact Δworkers=5 * scaleUpFactor=0.5) = 3
     # ROUND_UP(exact Δworkers=0.8 * scaleUpFactor=0.5) = 1
   else:
     actual Δworkers = ROUND_DOWN(exact Δworkers * scaleDownFactor)
     # examples:
     # ROUND_DOWN(exact Δworkers=-5 * scaleDownFactor=0.5) = -2
     # ROUND_DOWN(exact Δworkers=-0.8 * scaleDownFactor=0.5) = 0
     # ROUND_DOWN(exact Δworkers=-1.5 * scaleDownFactor=0.5) = 0
   ```

   A scaleUpFactor or scaleDownFactor of 1.0 means autoscaling will scale so that pending/available memory is 0 (perfect utilization).

3. Once the actual change to the number of workers is calculated, either the `scaleUpMinWorkerFraction` and `scaleDownMinWorkerFraction` acts as a threshold to

determine if autoscaling will scale the cluster. A small fraction signifies that autoscaling should scale even if the `Δworkers` is small. A larger fraction means that scaling should only occur when the `Δworkers` is large.

```
if (Δworkers >  scaleUpMinWorkerFraction* cluster size) then scale up
```

or

```
if (abs(Δworkers) >  scaleDownMinWorkerFraction* cluster size) then scale down
```

4. If the number of workers to scale is large enough to trigger scaling, autoscaling uses the `minInstances` `maxInstances` bounds of `workerConfig` and `secondaryWorkerConfig` and `weight` (ratio of primary to secondary workers) to determine how to split the number of workers across the primary and secondary worker instance groups. The result of these calculations is the final autoscaling change to the cluster for the scaling period.

Graceful decommissioning

Autoscaling supports YARN graceful decommissioning (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/GracefulDecommission.html) when removing nodes from a cluster. Graceful decommissioning allows applications to finish shuffling data between stages to avoid setting back job progress. The graceful decommission timeout (/dataproc/docs/concepts/configuring-clusters/scaling-clusters#graceful_decommissioning) provided in an autoscaling policy is the upper bound of the duration that YARN will wait for running applications (application that were running when decommissioning started) before removing nodes.

Autoscaling will not scale up during graceful decommissioning. It waits for an update operation, including gra
missioning, to complete, continues to wait for one cooldown period (for example, 2m), then issues another up
ion.

Multi-cluster policy usage

- An autoscaling policy defines scaling behavior that can be applied to multiple clusters. An autoscaling policy is best applied across multiple clusters when the clusters will share similar workloads or run jobs with similar resource usage patterns.
- You can update a policy that is being used by multiple clusters. The updates immediately affect the autoscaling behavior for all clusters that use the policy (see autoscalingPolicies.update (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies/update)). If you do not want a policy update to apply to a cluster that is using the policy, disable autoscaling on the cluster before updating the policy.

gcloud commandREST API (#rest-api)Console (#console)

Run the following `gcloud` command from a local terminal or in Cloud Shell (https://console.cloud.google.com/?cloudshell=true) to disable autoscaling on a cluster.

```
$ gcloud dataproc clusters update cluster-name --disable-autoscaling \
    --region=region
```

- A policy that is in use by one or more clusters cannot be deleted (see autoscalingPolicies.delete (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies/delete)).

Autoscaling recommendations

- **Cooldown period:** The minimum and default `cooldownPeriod` (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#basicyarnautoscalingconfig) is two minutes. If a shorter `cooldownPeriod` is set in a policy, workload changes will more quickly affect cluster size, but clusters may unnecessarily scale up and down. The recommended practice is to set a policy's `scale_up`, `scale_down`, and `min_worker_fractions` (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#basicautoscalingalgorithm)

to a non-zero value when using a shorter `cooldownPeriod`. This ensures that the cluster only scales up or down when the change in memory utilization is sufficient to warrant a cluster update.

- **Scaling Down:** MapReduce and Spark write intermediate shuffle data to local disk. Removing workers with shuffle data will delay the job's progress, as map tasks will need to be re-run to reproduce the shuffle data. Spark re-runs entire stages (https://issues.apache.org/jira/browse/SPARK-20178) if it detects that shuffle files are missing.
  - To scale down only when a cluster is idle, set `scale_down_factor` and `scale_down_min_worker_fraction` (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#basicyarnautoscalingconfig) to 1.0.
  - For *clusters with continuous load*, configure scaling down between jobs by setting scale_down (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#basicyarnautoscalingconfig) factor to 1.0 and setting a non-zero graceful_decommission_timeout (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies#basicyarnautoscalingconfig) to remove cluster nodes when they are not running YARN containers (see Graceful Decommissioning (/dataproc/docs/concepts/configuring-clusters/scaling-clusters#graceful_decommissioning)). Set the `graceful_decommission_timeout` to be longer than the longest-running cluster job to make sure that nodes will not be forcibly decommissioned before a job finishes.
  - Consider using Enhanced Flexibility Mode (/dataproc/docs/concepts/configuring-clusters/flex) to avoid or reduce the chance of losing job progress when removing nodes.
- **Spark jobs with *cached data*:** Set `spark.dynamicAllocation.cachedExecutorIdleTimeout` or uncaching datasets when they are no longer needed. By default Spark does not remove executors that have cached data.

Autoscaling with Apache Hadoop and Apache Spark

The following sections discuss how autoscaling does (or does not) interoperate with Hadoop YARN and Hadoop Mapreduce, and with Apache Spark, Spark Streaming, and Spark Structured Streaming.

Hadoop YARN Metrics

ın configure autoscaling for any YARN-based execution engine.

Autoscaling configures Hadoop YARN (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html) to schedule jobs based on YARN memory requests, not on YARN core requests.

Autoscaling is centered around the following Hadoop YARN metrics:

1. `Allocated memory` refers to the total YARN memory taken up by running containers across the whole cluster. If there are 6 running containers that can use up to 1GB, there is 6GB of allocated memory.
2. `Available memory` is YARN memory in the cluster not used by allocated containers. If there is 10GB of memory across all node managers and 6GB of allocated memory, there is 4GB of available memory. If there is available (unused) memory in the cluster, autoscaling may remove workers from the cluster.
3. `Pending memory` is the sum of YARN memory requests for pending containers. Pending containers are waiting for space to run in YARN. Pending memory is non-zero only if available memory is zero or too small to allocate to the next container. If there are pending containers, autoscaling may add workers to the cluster.

You can view these metrics in Cloud Monitoring (/monitoring/docs). As a default, YARN memory will be 0.8 * total memory on the cluster, with remaining memory reserved for other daemons and operating system use, such as the page cache. You can override the default value with the "yarn.nodemanager.resource.memory-mb" YARN configuration setting (see Apache Hadoop YARN, HDFS, Spark, and related properties (/dataproc/docs/concepts/configuring-clusters/cluster-properties#apache_hadoop_yarn_hdfs_spark_and_related_properties)).

Autoscaling and Hadoop MapReduce

MapReduce runs each map and reduce task as a separate YARN container. When a job begins, MapReduce submits container requests for each map task, resulting in a large spike in pending YARN memory. As map tasks finish, pending memory decreases.

When `mapreduce.job.reduce.slowstart.completedmaps` have completed (95% by default on Dataproc), MapReduce enqueues container requests for all reducers, resulting in another spike in pending memory.

Unless your map and reduce tasks take several minutes or longer, don't set a high value for autoscaling `scaleUpFactor`. Adding workers to the cluster takes at least 1.5 minutes, so make sure there is sufficient pending work to utilize new worker for several minutes. A good starting point is to set `scaleUpFactor` to 0.05 (5%) or 0.1 (10%) of pending memory.

Autoscaling and Spark

Spark adds an additional layer of scheduling on top of YARN. Specifically, Spark Core's dynamic allocation (https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation) makes requests to YARN for containers to run Spark executors, then schedules Spark tasks on threads on those executors. Dataproc clusters enable dynamic allocation by default, so executors are added and removed as needed.

Spark always asks YARN for containers, but without dynamic allocation, it only asks for containers at the beginning of the job. With dynamic allocation, it will remove containers, or ask for new ones, as necessary.

Spark starts from a small number of executors – 2 on autoscaling clusters – and continues to double the number of executors while there are backlogged tasks. This smooths out pending memory (fewer pending memory spikes). It is recommended that you set the autoscaling `scaleUpFactor` to a large number, such as 1.0 (100%), for Spark jobs.

Disabling Spark dynamic allocation

If you are running separate Spark jobs that do not benefit from Spark dynamic allocation, you can disable Spark dynamic allocation by setting `spark.dynamicAllocation.enabled=false` and setting `spark.executor.instances`. You can still use autoscaling to scale clusters up and down while the separate Spark jobs run.

Autoscaling and Spark Streaming

1. Since Spark Streaming has its own version of dynamic allocation (https://issues.apache.org/jira/browse/SPARK-12133) that uses streaming-specific signals to add and remove executors, set `spark.streaming.dynamicAllocation.enabled=true` and disable Spark Core's dynamic allocation by setting `spark.dynamicAllocation.enabled=false`.
2. Don't use Graceful decommissioning (/dataproc/docs/concepts/configuring-clusters/scaling-clusters#graceful_decommissioning) (autoscaling `gracefulDecommissionTimeout`) with Spark Streaming jobs. Instead, to safely remove workers with autoscaling, configure checkpointing (https://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing) for fault tolerance.

Alternatively, to use Spark Streaming without autoscaling:

1. Disable Spark Core's dynamic allocation (`spark.dynamicAllocation.enabled=false`), and
2. Set the number of executors (`spark.executor.instances`) for your job. See Cluster properties (/dataproc/docs/concepts/configuring-clusters/cluster-properties).

Autoscaling and Spark Structured Streaming

Autoscaling is not compatible with Spark Structured Streaming (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html) since Spark Structured Streaming currently does not support dynamic allocation (see SPARK-24815: Structured Streaming should support dynamic allocation (https://issues.apache.org/jira/browse/SPARK-24815)).

Controlling autoscaling through partitioning and parallelism

While parallelism is usually set or determined by cluster resources (for example, the number of HDFS blocks controls by the number of tasks), with autoscaling the converse applies: autoscaling sets the number of workers according to job parallelism. The following are guidelines to help you set job parallelism:

- While Dataproc sets the default number of MapReduce reduce tasks based on initial cluster size of your cluster, you can set `mapreduce.job.reduces` to increase the parallelism

of the reduce phase.

- Spark SQL and Dataframe parallelism is determined by `spark.sql.shuffle.partitions`, which defaults to 200.
- Spark's RDD functions default to `spark.default.parallelism`, which is set to the number of cores on the worker nodes when the job starts. However, all RDD functions that create shuffles take a parameter (https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism) for the number of partitions, which overrides `spark.default.parallelism`.

You should ensure your data is evenly partitioned. If there is significant key skew, one or more tasks may take significantly longer than other tasks, resulting in low utilization.

Autoscaling default Spark/Hadoop property settings

Autoscaling clusters have default cluster property values that help avoid job failure when primary workers are removed or secondary workers are preempted. You can override these default values when you create a cluster with autoscaling (see Cluster Properties (/dataproc/docs/concepts/configuring-clusters/cluster-properties)).

o increase job stability, you can create a cluster with non-preemptible secondary workers—for an example, see ptibles in a cluster
aproc/docs/concepts/compute/preemptible-vms?auto_signin=false#using_preemptibles_in_a_cluster).

Defaults to increase the maximum number of retries for tasks, application masters, and stages:

```
yarn.resourcemanager.am.max-attempts=10
d:mapreduce.map.maxattempts=10
d:mapreduce.reduce.maxattempts=10
:spark.task.maxFailures=10
:spark.stage.maxConsecutiveAttempts=10
```

Defaults to reset retry counters (useful for long-running Spark Streaming jobs):

```
:spark.yarn.am.attemptFailuresValidityInterval=1h
:spark.yarn.executor.failuresValidityInterval=1h
```

Default to have Spark's slow-start dynamic allocation (https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation) mechanism start from a small size:

```
:spark.executor.instances=2
```

Even though you may see shuffle fetch failures when primary workers are removed or secondary workers are pted, you should not increase shuffle fetch retries above their default values (**`spark.shuffle.io.maxRetri .shuffle.io.retryWaitMs=5000`** (5 seconds) for a total max retry duration value of 10 seconds). The reas files are lost, so additional retries only consume additional time.

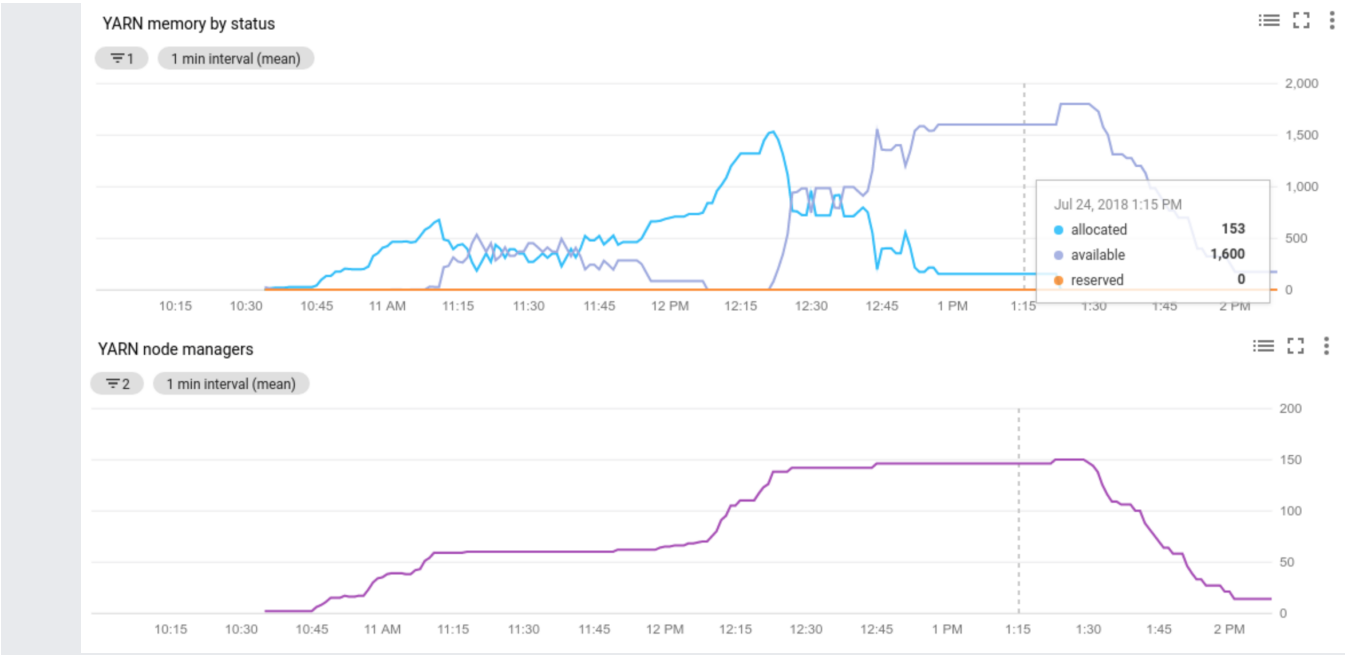
Autoscaling metrics and logs

The following resources and tools can help you monitor autoscaling operations and their effect on your cluster and its jobs.

Cloud Monitoring

Use Cloud Monitoring (/monitoring/docs) to:

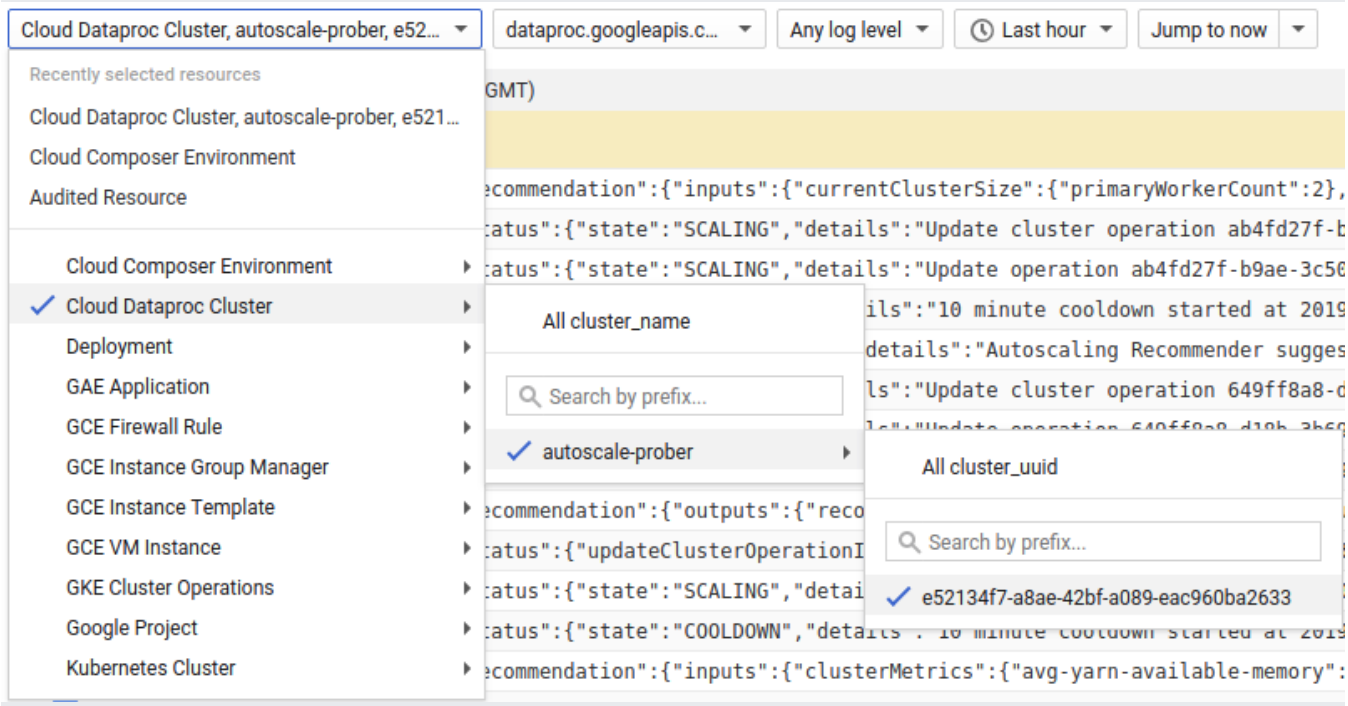
- view the metrics used by autoscaling
- view the number of Node Managers in your cluster
- understand why autoscaling did or did not scale your cluster



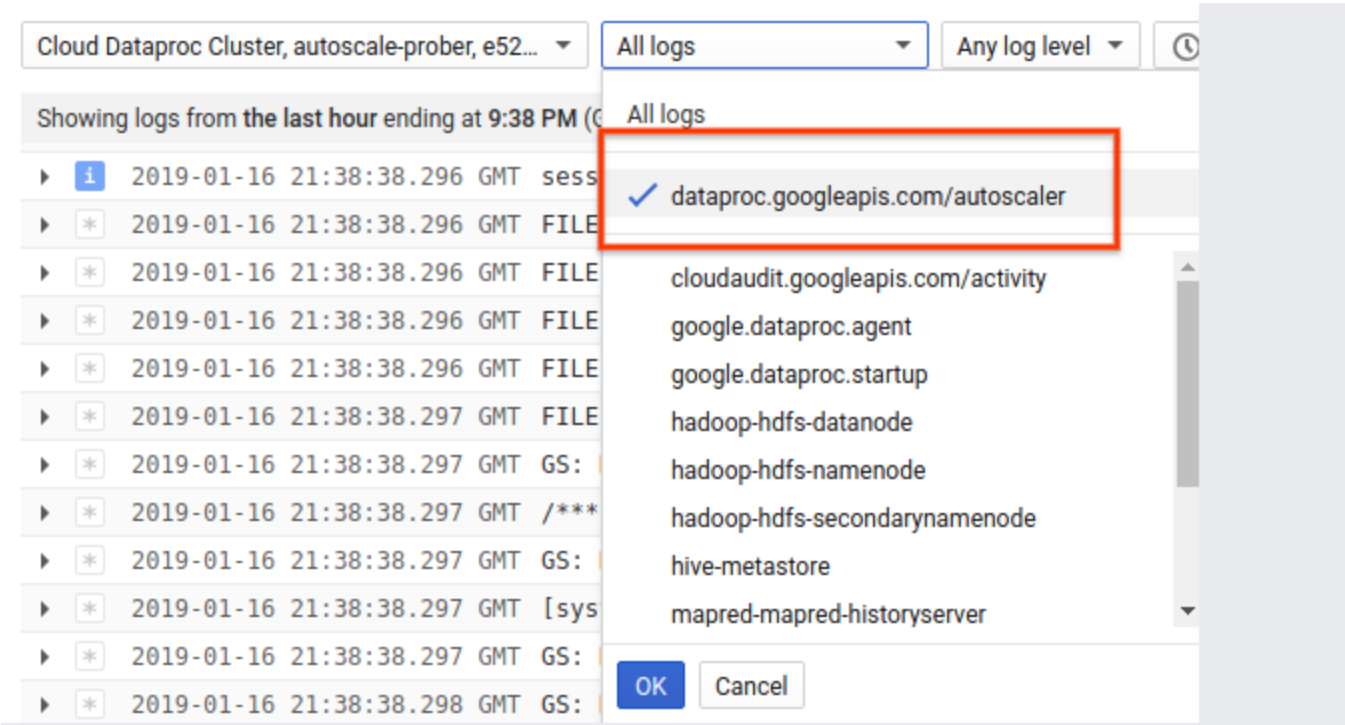
Cloud Logging

Use Cloud Logging (/logging) to view logs from the Cloud Dataproc Autoscaler.

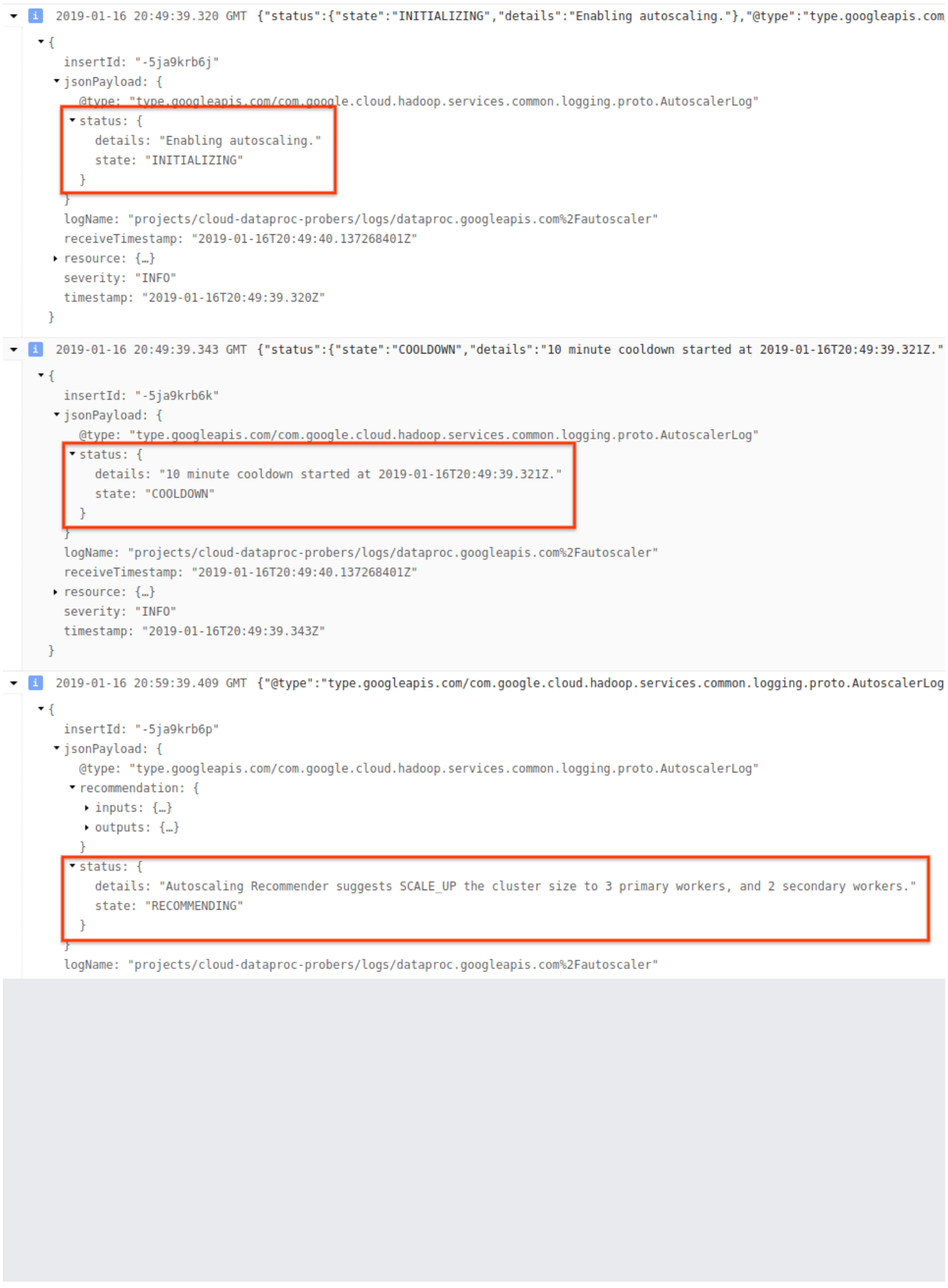
1) Find logs for your cluster.

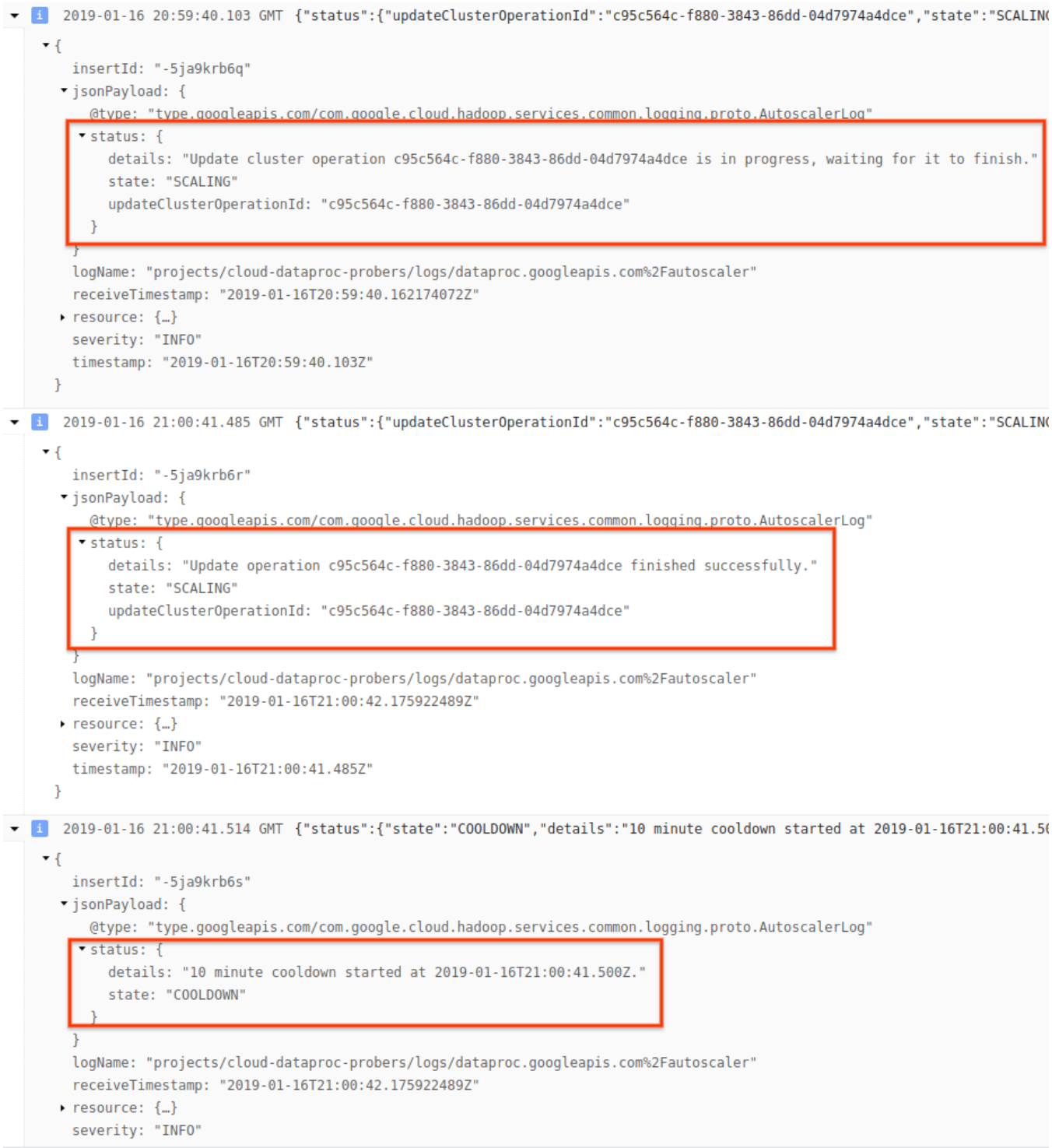


2) Select `dataproc.googleapis.com/autoscaler`.

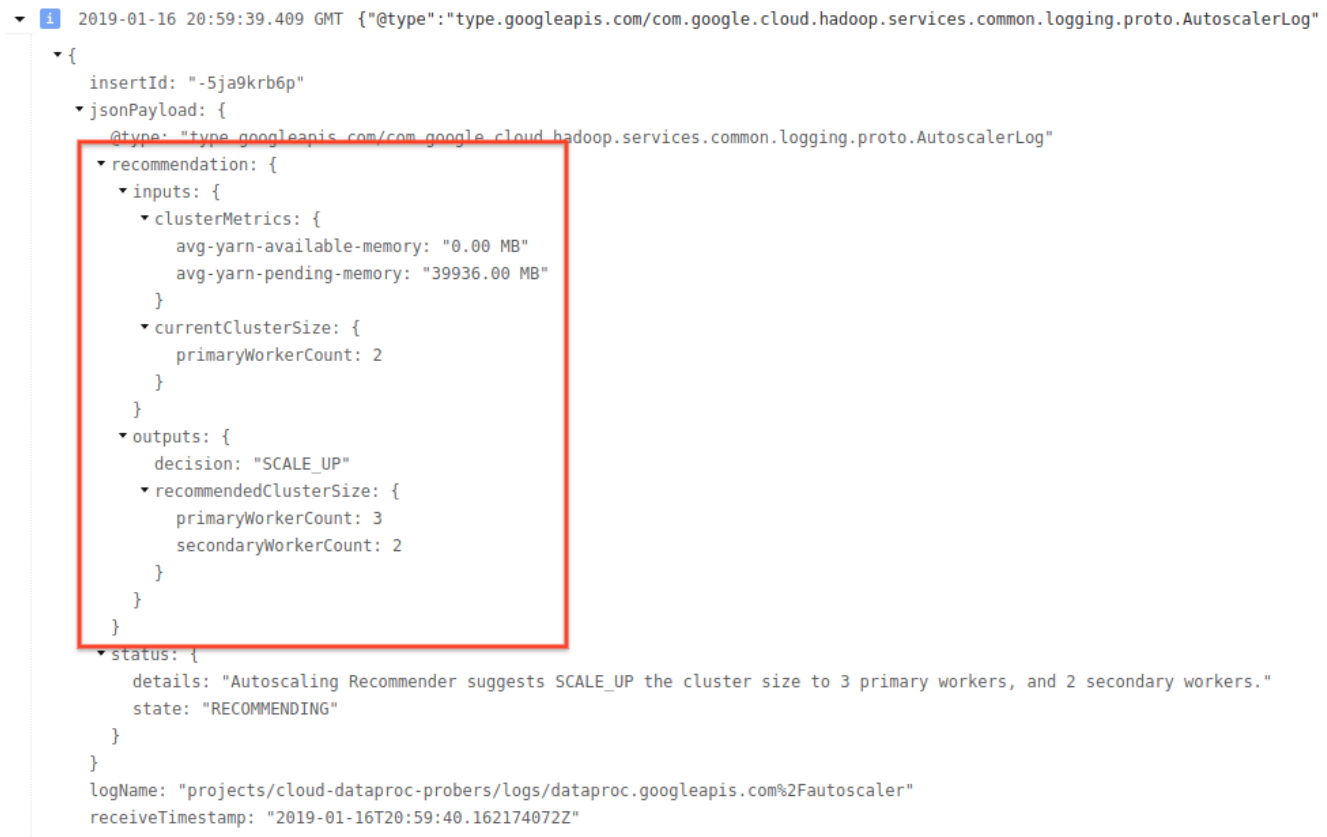


3) Expand the log messages to view the `status` field. The logs are in JSON, a machine readable format.


2019-01-16 20:49:39.320 GMT {"status":{"state":"INITIALIZING","details":"Enabling autoscaling."},"@type":"type.googleapis.com
{
insertId: "-5ja9krb6j"
jsonPayload: {
@type: "type.googleapis.com/com.google.cloud.hadoop.services.common.logging.proto.AutoscalerLog"
status: {
details: "Enabling autoscaling."
state: "INITIALIZING"
}
}
logName: "projects/cloud-dataproc-probers/logs/dataproc.googleapis.com%2Fautoscaler"
receiveTimestamp: "2019-01-16T20:49:40.137268401Z"
resource: {…}
severity: "INFO"
timestamp: "2019-01-16T20:49:39.320Z"
}
2019-01-16 20:49:39.343 GMT {"status":{"state":"COOLDOWN","details":"10 minute cooldown started at 2019-01-16T20:49:39.321Z."
{
insertId: "-5ja9krb6k"
jsonPayload: {
@type: "type.googleapis.com/com.google.cloud.hadoop.services.common.logging.proto.AutoscalerLog"
status: {
details: "10 minute cooldown started at 2019-01-16T20:49:39.321Z."
state: "COOLDOWN"
}
}
logName: "projects/cloud-dataproc-probers/logs/dataproc.googleapis.com%2Fautoscaler"
receiveTimestamp: "2019-01-16T20:49:40.137268401Z"
resource: {…}
severity: "INFO"
timestamp: "2019-01-16T20:49:39.343Z"
}
2019-01-16 20:59:39.409 GMT {"@type":"type.googleapis.com/com.google.cloud.hadoop.services.common.logging.proto.AutoscalerLog
{
insertId: "-5ja9krb6p"
jsonPayload: {
@type: "type.googleapis.com/com.google.cloud.hadoop.services.common.logging.proto.AutoscalerLog"
recommendation: {
inputs: {…}
outputs: {…}
}
status: {
details: "Autoscaling Recommender suggests SCALE_UP the cluster size to 3 primary workers, and 2 secondary workers."
state: "RECOMMENDING"
}
}
logName: "projects/cloud-dataproc-probers/logs/dataproc.googleapis.com%2Fautoscaler"



4) Expand the log message to see scaling recommendations, metrics used for scaling decisions, the original cluster size, and the new target cluster size.



Frequently Asked Questions (FAQs)

Can autoscaling be enabled on High Availability clusters and Single Node clusters?

Autoscaling can be enabled on High Availability clusters (/dataproc/docs/concepts/configuring-clusters/high-availability), but not on Single Node clusters (/dataproc/docs/concepts/configuring-clusters/single-node-clusters) (Single Node clusters do not support resizing).

Can you manually resize an autoscaling cluster?

Yes. You may decide to manually resize a cluster as a stop-gap measure when tuning an autoscaling policy. However, these changes will only have a temporary effect, and Autoscaling will eventually scale back the cluster.

Instead of manually resizing an autoscaling cluster, consider:

✓ Updating the autoscaling policy. **Any changes made to the autoscaling policy will affect all clusters that are currently using the policy** (see Multi-cluster policy usage (#multi-cluster_policy_usage)).

✓ Detaching the policy and manually scaling the cluster to the preferred size.

✓ Getting Dataproc support (/dataproc/docs/support/get-support).

What image versions support autoscaling? What API versions?

Autoscaling is supported through the v1 API (/dataproc/docs/reference/rest/v1/projects.locations.autoscalingPolicies) on cluster image versions 1.0.99+, 1.1.90+, 1.2.22+, 1.3.0+, and 1.4.0+ (see the Cloud Dataproc Version List (/dataproc/docs/concepts/versioning/dataproc-versions)) and through `gcloud dataproc autoscaling-policies` commands.

How is Dataproc different from Dataflow autoscaling?

See Comparing Cloud Dataflow autoscaling to Spark and Hadoop (/blog/search;query=Comparing%20Cloud%20Dataflow%20autoscaling%20to%20Spark%20and%20Hadoop;paginate=25;order=newest)

Last updated 2020-08-14 UTC.