Cloudera Streaming Analytics Operator 1.0.0

Flink Application Management

Date published: 2024-06-15
Date modified: 2024-06-15

CLOUDERA

https://docs.cloudera.com/

Legal Notice

Contents

Deploying Flink applications

Learn more about how to deploy Flink applications.

Procedure

1. Define the spec.job in the FlinkDeployment configuration file to create an application deployment as shown in the following example:

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkDeployment
metadata:
  name: flink-kubernetes-tutorial
spec:
  image: [***REGISTRY HOST***]:[***PORT***]/[***PROJECT***]/flink-kuber
netes-tutorial:latest
  flinkVersion: v1_18
  flinkConfiguration:
    taskmanager.numberOfTaskSlots: "4"
  serviceAccount: flink
  mode: native
  jobManager:
    resource:
      memory: "2048m"
      cpu: 1
  taskManager:
    resource:
      memory: "2048m"
      cpu: 1
  job:
    args: ["--rowsPerSec", "10"]
    jarURI: local:///opt/flink/usrlib/flink-kubernetes-tutorial.jar
    parallelism: 4
    state: running
    upgradeMode: stateless
```

The following properties are required for the application deployment:

Property	Description
jarURI	URI of the Flink job JAR file in the Docker image.
parallelism	Parallelism of the Flink job.
state	State of the Flink job that can be running or suspended.
upgradeMode	Upgrade mode of the Flink job that can be one of the following: • stateless: No state will be saved • last-state: Uses Flink high availability metadata to resume jobs • savepoint: Uses Flink savepoints to cancel and resume jobs

2. Submit the YAML file to run the application when configuring the `FlinkDeployment` resource is complete:

```
kubectl apply -f your-deployment.yaml
```

Job management

Learn more about Flink job management.

In case you make any changes to the `FlinkDeployment` resource that requires a restart, the Flink Operator automatically restarts the deployment after applying a patch to the resource. For example, changing the job arguments can be done with the following:

```
kubectl -n flink patch FlinkDeployment flink-kubernetes-tutorial \
    --type=merge \
    --patch='{"spec":{"job":{"args":["--rowsPerSec", "100"]}}}'
```

In this example, patch is used as an alternative to modify the original configuration and apply the changes to the `FlinkDeployment` resource.

To restart the deployment without making any changes to the definition, you can update the spec.restartNonce property. This ensures that the Flink Operator automatically restarts the job if it is different from the previous value.

```
kubectl -n flink patch FlinkDeployment flink-kubernetes-tutorial \
    --type=merge \
    --patch='{"spec":{"restartNonce":1234}}'
```

Recovering missing job deployments

In case the Flink cluster deployment is deleted by accident or external process, the Flink Operator can recover the deployment when high availability is enabled. Ensure that the kubernetes.operator.jm-deployment-recovery.enabled property is enabled to recover the `FlinkDeployment`.

Restarting unhealthy job deployments

In case the Flink cluster deployment is considered unhealthy, the Flink Operator can restart the deployment when high availability is enabled. Ensure that the following properties are enabled to restart the Flink deployment:

- kubernetes.operator.cluster.health-check.enabled
- kubernetes.operator.jm-deployment-recovery.enabled

A Flink deployment is considered unhealthy in the following cases:

- The count of Flink restarts reaches the configured value (default is 64) for kubernetes.operator.cluster.health-check.restarts.threshold property within the window period (default is 2 minutes) configured for kubernetes.operator.cluster.health-check.restarts.window.

If cluster.health-check.checkpoint-progress.enabledis enabled and the count of successful Flink checkpoints do not change within the window period (default is 5 minutes) configured for kubernetes.operator.cluster.health-check.checkpoint-progress.window

Restarting failed job deployments

In case the Flink job is failed, the Flink Operator can restart the failed job when kubernetes.operator.job.restart.failed property is enabled. In this case when the job status is FAILED the Flink Operator deletes the current job and redeploys it using the latest successful checkpoint.

Manually recovering deployments

In case the Flink deployment is in a state where the Flink Operator cannot determine the health of the application or the latest checkpoint cannot be used to recover the deployment, manual recovery can be used.

You have the following options to restore a job from the target savepoint or checkpoint:

Redeploying with savepointRedeployNonce

You can redeploy a Flink Deployment or Flink Session Job resource from a target savepoint by using the savepointRedeployNonce andinitialSavepointPath in thejob.spec as shown in the following example:

```
job:
```

```
    initialSavepointPath: file://redeploy-target-savepoint
    # If not set previously, set to 1, otherwise increment, e.g. 2
    savepointRedeployNonce: 1
```

When changing the savepointRedeployNonce the operator will redeploy the job to the savepoint defined in the initialSavepointPath. The savepoint path must not be empty.

Deleting and recreating resources

You also have the option to completely delete and recreate the resources to solve any deployment related issues. This resets the status information to start from a clean slate. However, savepoint history will be lost and the Flink Operator will not clean up past periodic savepoints taken before the deletion. You can use the following steps to recreate the `FlinkDeployment` resource from a user defined savepoint path:

1. Locate the latest checkpoint or savepoint metafile in the configured checkpoint or savepoint directory.
2. Delete the `FlinkDeployment` resource of your application.
3. Check that the current savepoint is still present, and that your `FlinkDeployment` resource is deleted completely.
4. Modify the job.spec and set the initialSavepointPath to the last checkpoint location.
5. Recreate the `FlinkDeployment` resource.
6. Monitor the job to see what caused the problem before.

Job lifecycle management

Learn more about Flink job lifecycle management.

You can control the state of the application using the state property of the spec.job in the `FlinkDeployment` resource. The following application states are supported:

- running: The job is expected to be running and processing data.
- suspended: Data processing is temporarily suspended, with the intention of continuing later.

You can stop the Flink job by modifying the spec.job.state from running to suspended.

```
$ kubectl -n [*** NAMESPACE ***] patch FlinkDeployment [*** FLINK DEPLOYMENT
 NAME ***] \
    --type=merge \
    --patch='{"spec":{"job":{"state":"suspended"}}}'
```

Suspended jobs can be restarted using the same method:

```
$ kubectl [*** NAMESPACE ***] patch FlinkDeployment [*** FLINK DEPLOYMENT
 NAME ***] \
    --type=merge \
    --patch='{"spec":{"job":{"state":"running"}}}'
```

The following state transition scenarios exist when updating the existing Flink Deployment resource:

- from running to running: Job upgrade operation. In practice, a suspend followed by a restore operation.
- from running to suspended: Suspend operation to stop the application while maintaining the application state.
- from suspended to running: Restore operation to start the application from current state using the latest spec.
- from suspended to suspended : Deployment spec is updated, but the application is not started.

The explained state changes do not remove the `FlinkDeployment` resource from the cluster, the operation is simply suspended. When you no longer wish to process data using an existing `FlinkDeployment` resource, the following command can be used to delete the application:

```
kubectl -n [*** NAMESPACE ***] delete FlinkDeployment [*** FLINK DEPLOYMENT
 NAME ***]
```

Application upgrades

Learn more about Flink application upgrades.

When the job specifications are changed for a `FlinkDeployment` or `FlinkSessionJob` resource, the running application must be upgraded. In case of upgrades, the Flink Operator automatically stops the currently running application, if it's not in a suspended state. After stopping, the Flink Operator redeploys the application using the new specification. When redeploying stateful applications, their state is carried over from (suspended remains suspended, running will be started again).

You can configure how states are managed when stopping and restarting stateful applications using the upgradeMode setting in spec.job. The following values are supported for upgradeMode:

- stateless: stateless application upgrades from empty state
- savepoint: a savepoint is created during the upgrade process to provide safety and possibility for the savepoint to be used as backup. The Flink application must be in running state to allow the savepoint to be created. In case the application is in an unhealthy state, the last checkpoint will be used, unless kubernetes.operator.job.upgrade.last-state-fallback.enabled is set to false. If the last checkpoint is not available, the job upgrade will fail. For more information, see *Savepoint management*.
- last-state: the latest checkpoint information is used for quick upgrades in any application state (even for failing jobs). Healthy application state is not required as the latest checkpoint information is used. Manual recovery might be necessary in case the high availability metadata is lost. You can configure the kubernetes.operator.job.upgrade.last-state.max.allowed.checkpoint.age to limit the time the application may fall back to when picking up the latest checkpoint. If the checkpoint is older than the configured value, a savepoint will be created instead (for healthy applications only).

Note: The last-state value for upgradeMode is not supported for session clusters.

The upgradeMode configuration controls both the stop and restore mechanisms as shown in the following table:

Table 1:

	Stateless	Last state	Savepoint
Configuration Requirement	None	Checkpointing & HA Enabled	Checkpoint/Savepoint directory defined
Job Status Requirement	None	HA metadata available	Job Running[1]
Suspend Mechanism	Cancel/Delete	Delete Flink deployment (keep HA metadata)	Cancel with savepoint
Restore Mechanism	Deploy from empty state	Recover last state using HA metadata	Restore From savepoint
Production Use	Not recommended	Recommended	Recommended

[1] When HA is enabled and the application is in an unhealthy state, the savepoint upgrade mode might fall back to the last-state behavior.

Related Information

Savepoint management

Savepoint management

Learn more about Flink savepoint management.

Savepoints are triggered automatically by the system during the upgrade process, as described in the previous section. You can also trigger savepoints manually or periodically, but user-created savepoints will not be used during the restoration process after the upgrade, and are not required for correct operation.

For savepoints to work, Flink requires a durable storage to save its data. You can use any type of (local or networked) mounted volumes, or object storage (for example S3, Longhorn, NFS, etc). In this documentation we use an NFS volume type.

To enable and use savepoints, you need to update the following properties (compared to the previous specifications):

- Define a new volume to store the savepoint and mount it to the flink-main-container container.
- Enable savepoints by adding the savepoint directory to spec.flinkConfiguration.
- Enable checkpoints by adding the checkpoint directory to spec.flinkConfiguration.
- Enable periodic savepoints triggered by the Flink Operator by adding kubernetes.operator.periodic.savepoint.interval: 2h.
- Set upgradeMode to savepoint to create savepoints and resume from them before each restart.

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkDeployment
metadata:
  name: flink-kubernetes-tutorial
spec:
  image: [***REGISTRY HOST***]:[***PORT***]/[***PROJECT***]/flink-kubernete
s-tutorial:latest
  flinkVersion: v1_18
  flinkConfiguration:
    taskmanager.numberOfTaskSlots: "4"
    state.savepoints.dir: file:///opt/flink/durable/savepoints
    state.checkpoints.dir: file:///opt/flink/durable/checkpoints
    high-availability.storageDir: file:///opt/flink/durable/ha
    kubernetes.operator.periodic.savepoint.interval: 2h
  serviceAccount: flink
  mode: native
  jobManager:
    resource:
      memory: "2048m"
      cpu: 1
  taskManager:
    resource:
      memory: "2048m"
      cpu: 1
  podTemplate:
    spec:
      containers:
        - name: flink-main-container
          volumeMounts:
            - mountPath: /opt/flink/durable
              name: flink-volume
      volumes:
        - name: flink-volume
          nfs:
            server: my-nfs-server.example.com
```

```
          path: /data/flink/
  job:
    args: ["--rowsPerSec", "10", "--outputPath", "/opt/flink/durable"]
    jarURI: local:///opt/flink/usrlib/flink-kubernetes-tutorial.jar
    parallelism: 4
    state: running
    upgradeMode: savepoint
```

You can use the following commands to create the new deployment:

```
kubectl -n flink delete FlinkDeployment flink-kubernetes-tutorial
kubectl -n flink apply -f flink-deployment.yaml
```

After the application is running, you trigger a savepoint using the following command:

```
kubectl -n flink patch FlinkDeployment flink-kubernetes-tutorial \
    --type=merge \
    --patch='{"spec":{"job":{"savepointTriggerNonce":1234}}}'
```

In case the application is suspended, the Flink Operator automatically creates a savepoint and resumes the application from the savepoint when restarted.

The Flink Operator automatically keeps track of the savepoint history, whether it's triggered automatically by an upgrade or manually (ad-hoc or by a periodic task). You can configure an automatic removal of older savepoints by changing the cleanup behavior as shown in the following example:

```
kubernetes.operator.savepoint.history.max.age: 24 h
kubernetes.operator.savepoint.history.max.count: 5
```

You can disable the savepoint cleanup completely by setting the kubernetes.operator.savepoint.cleanup.enabled property to false. In this case, the Flink Operator still collects and saves the savepoint history, but does not perform any cleanup operations.

Additional savepoint operations

Even though savepoints are triggered automatically during an upgrade process, you can also trigger a savepoint manually or periodically. These configurations are optional and have no impact on the automatic savepoint triggering, and not required for the correct operation of the Flink cluster.

Manually triggering a savepoint

You can use the savepointTriggerNonce property in spec.job to create a new savepoint by defining a new (different or random) value to the property:

```
job:
    ...
    savepointTriggerNonce: 123
```

This change will be applied by the Flink Operator as described in the previous sections.

Periodically triggering a savepoint

You can use the kubernetes.operator.periodic.savepoint.interval property, on a per-job level, to trigger a savepoint after a specified period:

```
flinkConfiguration:
    ...
    kubernetes.operator.periodic.savepoint.interval: 6h
```

The timely execution of the periodic savepoint is not guaranteed as it can be delayed due to unhealthy job status or other user operation.

Routing with ingress

Learn more about routing with ingress.

The Flink Operator supports creating Ingress entries for external User Interface (UI) access. The Ingress solution is ideal for production environments, and the manual port-forwarding of the service port can be used for smaller local jobs.

Ingress controllers allow you to route traffic from outside the Kubernetes cluster to your Service resources by providing a single point of entry and routing the traffic based on the data in the request (for example, URL path) to the correct services. Ingress can also be used to easily set up HTTPS for your services without the need to install any certificates to Flink itself.

Note: Before deploying the Flink Deployment resource using ingress, ensure that the NGINX Ingress controller is installed on your Kubernetes cluster. If you have an OpenShift cluster, then you might already have HAProxy enabled, and that will automatically pick up new Ingress resources created by the operator.

To use the Ingress controller, you must create the Ingress resources in the Kubernetes cluster with the required filters and configurations that describe when and how to route requests to the Flink service. This can be done by adding the spec.ingress the `FlinkDeployment` resource as shown in the following example:

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkDeployment
metadata:
  name: flink-kubernetes-tutorial
spec:
  image: [***REGISTRY HOST***]:[***PORT***]/[***PROJECT***]/flink-kubernete
s-tutorial:latest
  flinkVersion: v1_18
  flinkConfiguration:
    taskmanager.numberOfTaskSlots: "4"
  serviceAccount: flink
  mode: native
  jobManager:
    resource:
      memory: "2048m"
      cpu: 1
  taskManager:
    resource:
      memory: "2048m"
      cpu: 1
  job:
    jarURI: local:///opt/flink/usrlib/flink-kubernetes-tutorial.jar
    parallelism: 4
    state: running
    upgradeMode: stateless
  ingress:
    className: nginx
    template: "[***HOSTNAME***]/{{namespace}}/{{name}}(/|$)(.*)"
    annotations:
      nginx.ingress.kubernetes.io/rewrite-target: "/$2"
```

You can use the following command to create the new deployment:

```
kubectl -n flink apply -f flink-deployment.yaml
```

The Flink Operator will automatically create the `Ingress` resources specified when creating the deployment. If you inspect the newly created `Ingress` resource, it should look something like this:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
 nginx.ingress.kubernetes.io/rewrite-target: /$2
  name: flink-kubernetes-tutorial
  namespace: flink
spec:
  rules:
  - http:
   paths:
   - backend:
       service:
         name: flink-kubernetes-tutorial-rest
         port:
           number: 8081
     path: /flink/flink-kubernetes-tutorial(/|$)(.*)
     pathType: ImplementationSpecific
```

You can see that the Operator has replaced the template /{{namespace}}/{{name}}(/|$)(.*) with /flink/flink-kubernetes-tutorial(/|$)(.*) which corresponds to the namespace and name of the job. This makes it easier to run multiple jobs with the same ingress configuration, even in multiple namespaces.

You can also notice that two Regex capturing groups are specified in the path filter. The nginx.ingress.kubernetes.io/rewrite-target annotation instructs the Ingress controller to rewrite the URI path to only contain characters matched by the second capture group (in this example, (.*)).

This will re-write the path of http://localhost/flink/flink-kubernetes-tutorial/#/job/running to simply be /#/job/running when routing it to the Flink service.

You can further customize it using the template template: "flink.mydomain.com/{{namespace}}/{{name}}(/|$)(.*)". This will add the host flink.mydomain.com to the rules list and allows for even greater freedom of configuration.

Note: In case you use HAProxy (which is the default on OpenShift), you might need to change some configurations, as shown in the following examples:

```
ingress:
  template: "[***INGRESS FQDN***]/{{namespace}}/{{name}}"
  annotations:
    haproxy.router.openshift.io/rewrite-target: /
```

Sidecars with pod template

You can extend your FlinkDeployment in case you want to add more containers in your Kubernetes pod using the pod template and sidecars.

The Flink Operator CRD has a minimal set of settings to express the basic attributes of a deployment. For more customization you can use the flinkConfiguration and podTemplate properties.

Pod templates allow customization of the Flink job and task manager pods, to, for example, specify volume mounts, ephemeral storage, sidecar containers and so on.

Pod templates can be layered as shown in the below example. You can define the settings for the pod templates to be applied to both the job and task manager in a common pod template. You can also add another template under the job or task manager to define additional settings that supplement (or override) the common template, for example when using sidecars.

Defining sidecars instruct the Flink Operator to create other containers in the Flink JobManager and TaskManager pods, for example:

- to download artifacts (for example, JAR files) before executing the job
- to collect metrics and logs from Flink during runtime and analyze/save them.

The following example sets up another container running next to Flink in all the created pods to periodically output the size of the log file:

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkDeployment
metadata:
  name: flink-kubernetes-tutorial
spec:
  image: [***REGISTRY HOST***]:[***PORT***]/[***PROJECT***]/flink-kubernete
s-tutorial:latest
  flinkVersion: v1_18
  flinkConfiguration:
    taskmanager.numberOfTaskSlots: "4"
  serviceAccount: flink
  mode: native
  jobManager:
    resource:
      memory: "2048m"
      cpu: 1
  taskManager:
    resource:
      memory: "2048m"
      cpu: 1
  job:
    jarURI: local:///opt/flink/usrlib/flink-kubernetes-tutorial.jar
    parallelism: 4
    state: running
    upgradeMode: stateless
  podTemplate:
    spec:
      containers:
        - name: flink-main-container
          volumeMounts:
            - mountPath: /opt/flink/log
              name: flink-logs
        - name: sidecar
          image: busybox
          command: [ 'sh','-c','while true; do wc -l /flink-logs/*.log; s
leep 5; done' ]
          volumeMounts:
            - mountPath: /flink-logs
              name: flink-logs
      initContainers:
        # Sample sidecar container
        - name: sidecar-init
          image: busybox
          command: [ 'sh', '-c', 'echo initContainer loaded' ]
      volumes:
        - name: flink-logs
          emptyDir: {}
```

You can use the following commands to create the new deployment:

```
kubectl -n flink apply -f flink-deployment.yaml
```

This sidecar creates a new temporary volume called flink-logs in the Flink main container that is mounted to the default log output path, /opt/flink/log. The example also creates a BusyBox sidecar that also mounts the same volume and periodically prints the logs' line count.

Note: You must use the flink-main-container name to modify the Flink container, so the Flink Operator can merge the configurations together when creating the container.

The init-container is a type of container that needs to finish running and exit with code 0 before the other containers can start. As an example, this can be used to download artifacts for the Flink jobs.

Autoscaler

The Flink Operator offers a job autoscaler functionality that can scale individual job vertices (chained operator groups) based on various metrics collected from running Flink applications.

Note: The term "operator" in this section refers to the function of the Flink Operator to transform one or more DataStreams into a new DataStream. For more information, see *DataStream operators* documentation.

The autoscaler can be used to eliminate back pressure and satisfy a set utilization target. Adjusting the parallelism for a job on vertex level enables efficient autoscaling of complex and heterogeneous streaming applications. The autoscaler uses the built-in job upgrade mechanism to perform the rescaling.

The autoscaler has the following key benefits:

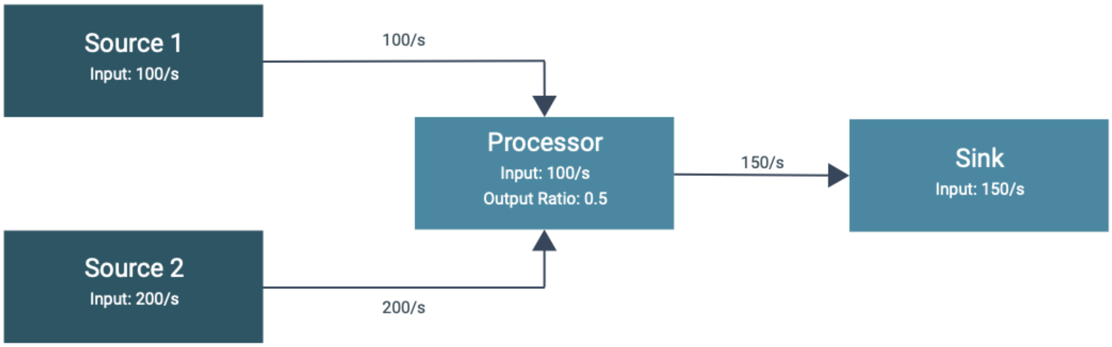
- Better cluster resource utilization and lower operating costs
- Automatic parallelism tuning for even complex streaming pipelines
- Automatic adaptation to changing load patterns
- Detailed utilization metrics for performance debugging

The autoscaler uses the metrics exposed by the Flink metric system. The following metrics are collected directly from a Flink job:

- Backlog information at each source
- Incoming data rate at the sources (for example, records per sec written into a Kafka topic)
- Number of records processed per second in each job vertex
- Busy time per second of each job vertex (current utilization)

Note: Container memory and CPU utilization metrics are not used by the autoscaler directly. High utilization is reflected in the processing rate and busy time metrics of the individual job vertices.

The autoscaler algorithm calculates the required processing capacity and target data rate for each operator starting from the source. The target data rate for the source vertices is equal to the incoming data rate. For downstream operators, the target data rate is calculated as the sum of the input (upstream) operators output data rate along the given edge in the processing graph.



The target utilization percentage of the operators can be configured in the pipeline. For example, you can keep all operators busy between 60% and 80%. The autoscaler will find a parallelism configuration that matches the output rates of all operators with the input rates of all downstream operators at the targeted utilization. As the load increases or decreases, the autoscaler adjusts the parallelism levels of the individual operator to fulfill the current rate over time.

Note: Before using the autoscaler, ensure that you met all the necessary requirements and you are aware of the limitations. For more information, see *Autoscaler limitations* page.

Related Information

DataStream operators | Apache Flink

Autoscaler limitations | Apache Flink Kubernetes Operator

Autoscaler configurations

Learn more about how to configure the autoscaler.

You can tune the autoscaler by changing the default configurations based on your environment:

```
...
flinkVersion: v1_18
flinkConfiguration:
    job.autoscaler.enabled: "true"
    job.autoscaler.stabilization.interval: 1m
    job.autoscaler.metrics.window: 5m
    job.autoscaler.target.utilization: "0.6"
    job.autoscaler.target.utilization.boundary: "0.2"
    job.autoscaler.restart.time: 2m
    job.autoscaler.catch-up.duration: 5m
    pipeline.max-parallelism: "720"
```

You can use the following configurations to change the behavior of the autoscaler:

Table 2: Autoscaler configuration properties

Configuration	Default value	Description
job.autoscaler.enabled	false	Enables or disables the autoscaler functionality. The default false value still supports a passive/metrics-only mode. In this case the autoscaler only collects and evaluates scaling related performance metrics, but does not trigger any job upgrades. This can be used to learn using the autoscaler without any impact on the running applications.

Configuration	Default value	Description
job.autoscaler.stabilization.interval	5 minutes	Specifies the stabilization period in which no new scaling will be executed.
job.autoscaler.metrics.window	15 minutes	Specifies the size of the scaling metrics aggregation window. The size of the window determines how small fluctuations affect the autoscaler: more stability can be achieved with increased window size, but with larger windows the autoscaler might be slower to react to sudden changes.
job.autoscaler.target.utilization	0.7	Specifies the target vertex utilization for stable job performance and some buffer for load fluctuations. The default 0.7 targets 70% utilization/load for the job vertexes.
job.autoscaler.target.utilization.boundary	0.3	Specifies the target of vertex utilization boundary for an extra buffer to avoid immediate scaling on load fluctuations. The default 0.3 targets 30% deviation from the target utilization before triggering a scaling action.
job.autoscaler.restart.time	5 minutes	Specifies the expected time an application restarts.
job.autoscaler.catch-up.duration	30 minutes	Specifies the expected time to entirely process any backlog after a scaling operation is completed. When lowering the catch-up duration, the autoscaler reserves more extra capacity for the auto scaling actions.
pipeline.max-parallelism	200	Specifies the maximum parallelism the autoscaler can use. This limit is ignored if the value is higher than the max parallelism configured in the Flink configuration or directly on each operator. To ensure flexible scaling, it is recommended to choose max parallelism configurations that have a lot of divisors, such as 120, 180, 240, and so on.

For the full list of configuration properties, see the *Autoscaler configuration* page.

Related Information

Autoscaler Configuration | Apache Flink Kubernetes Operator