# Batches API

| Verb | Description |
| --- | --- |
| Spark.batches.describe() | [Describe batch pipelines across a tenant](#describe-batch-pipelines-across-a-tenant). |
| Spark.batches.create(params, [options]) | [Create a new batch pipeline](#create-a-new-batch-pipeline). |
| Spark.batches.of(id) | [Define a client-side batch pipeline by ID](#X1fd4cfc07f99b01679ba60e1102d5f09fb28047). |
| Spark.batches.of(id).get\_info() | [Get the details of a batch pipeline](#get-the-details-of-a-batch-pipeline). |
| Spark.batches.of(id).get\_status() | [Get the status of a batch pipeline](#get-the-status-of-a-batch-pipeline). |
| Spark.batches.of(id).push(data, [options]) | [Add input data to a batch pipeline](#add-input-data-to-a-batch-pipeline). |
| Spark.batches.of(id).pull([options]) | [Retrieve the output data from a batch pipeline](#X5dbbb5c5ccc2e34566996ab904c5f7fa2602309). |
| Spark.batches.of(id).dispose() | [Close a batch pipeline](#close-a-batch-pipeline). |
| Spark.batches.of(id).cancel() | [Cancel a batch pipeline](#cancel-a-batch-pipeline). |

The [Batches API](https://docs.coherent.global/spark-apis/batch-apis) (in Beta) offers a set of endpoints that facilitate the execution of a Spark service for handling a large volume of input data. Spark provides a dedicated infrastructure that is specifically designed for parallel processing. The infrastructure is capable of scaling up or down based on the data volume that needs to be processed.

By utilizing this API, you will ensure optimal performance and scalability for your data processing tasks.

[!NOTE] It is important to note that the Batches API is recommended when dealing with datasets consisting of more than 10,000, with a calculation time of about 500ms or more. Unless you have specific requirements or reasons to use a different approach, such as [Services API](./services.md) as an alternative, this API is the way to go.

For more information on the Batches API and its endpoints, refer to the [API reference](https://docs.coherent.global/spark-apis/batch-apis).

## Describe batch pipelines across a tenant

This method returns a list of all the batch pipelines that are available across a tenant. It helps you keep track of your batch pipelines and their statuses. Keep in mind that this will only provide information about batches that are in progress or recently completed (i.e., within the past hour).

spark.batches.describe()

### Arguments

This method does not require any arguments.

### Returns

The method returns a list of batch pipelines along with their details. You will only retrieve information about batches that were initiated by your user account unless you have been granted access (e.g., supervisor:pf) to view other users’ batches.

{  
 "in\_progress\_batches": [],  
 "recent\_batches": [  
 {  
 "object": "batch",  
 "id": "uuid",  
 "data": {  
 "pipeline\_status": "closed",  
 "summary": {  
 "records\_submitted": 123,  
 "records\_failed": 0,  
 "records\_completed": 123,  
 "compute\_time\_ms": 13,  
 "batch\_time\_ms": 456  
 },  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "completed",  
 "created\_by": "john.doe@coherent.global",  
 "created\_timestamp": "1970-12-03T04:56:12.186Z",  
 "updated\_timestamp": "1970-12-03T04:57:12.186Z",  
 "service\_uri": "my-folder/my-service[0.4.2]"  
 }  
 }  
 ],  
 "tenant": {  
 "configuration": {  
 "input\_buffer\_allocated\_bytes": 0,  
 "output\_buffer\_allocated\_bytes": 0,  
 "max\_workers": 100  
 },  
 "status": {  
 "input\_buffer\_used\_bytes": 0,  
 "input\_buffer\_remaining\_bytes": 0,  
 "output\_buffer\_used\_bytes": 0,  
 "output\_buffer\_remaining\_bytes": 0,  
 "workers\_in\_use": 0  
 }  
 },  
 "environment": { "update": 123 }  
}

## Create a new batch pipeline

This method allows you to start a new batch pipeline. This is a necessary step before you can start performing any operations on the batch pipeline.

[!IMPORTANT] It is a good practice to retain the id of the newly-created batch pipeline. This identifier will be used to reference the batch pipeline in subsequent operations.

### Arguments

The method accepts a string or a UriParams object and optional keyword arguments, which include metadata and other pipeline configuration settings (experimental).

For the first argument, the service URI locator as a string or UriParams object:

| Property | Type | Description |
| --- | --- | --- |
| *folder* | None \| str | The folder name. |
| *service* | None \| str | The service name. |
| *version* | None \| str | The user-friendly semantic version of a service. |
| *version\_id* | None \| str | The UUID of a particular version of the service. |
| *service\_id* | None \| str | The service UUID (points to the latest version). |

spark.batches.create('my-folder/my-service')  
# or  
spark.batches.create(UriParams(folder='my-folder', service='my-service'))

If needed, you can also provide additional keyword arguments to configure how the batch pipeline, once created, will perform its operations.

| Property | Type | Description |
| --- | --- | --- |
| *active\_since* | None \| str | The transaction date (helps pinpoint a version). |
| *source\_system* | None \| str | The source system (defaults to Spark Python SDK). |
| *correlation\_id* | None \| str | The correlation ID. |
| *call\_purpose* | None \| str | The call purpose. |
| *subservices* | None \| str \| List[str] | The list of subservice to output. |
| *selected\_outputs* | None \| str \| List[str] | Select which output to return. |
| *unique\_record\_key* | None \| str \| List[str] | Indicate certain inputs as unique identifiers. |

The following optional arguments are experimental and may change in future releases.

| Property | Type | Description |
| --- | --- | --- |
| *min\_runners* | None \| int | Number of concurrent runners used to start a batch in a VM before ramping up (defaults to 10). |
| *max\_runners* | None \| int | Maximum number of concurrent runners allowed in a VM (defaults to 100). |
| *chunks\_per\_vm* | None \| int | Number of chunks to be processed by all VMs (defaults to 2). |
| *runners\_per\_vm* | None \| int | Number of runners per VM (defaults to 2). |
| *max\_input\_size* | None \| float | Maximum input buffer (in MB) a batch pipeline can support. |
| *max\_output\_size* | None \| float | Maximum output buffer (in MB) a batch pipeline can support. |
| *accuracy* | None \| float | Acceptable error rate between 0.0 - 1.0 (defaults to 1.0 aka 100%). |

### Returns

The method returns a dictionary containing the details of the newly created batch pipeline. Do note the schema is similar to the one returned by the [dispose method](#close-a-batch-pipeline) or the [cancel method](#cancel-a-batch-pipeline).

{  
 "object": "batch",  
 "id": "uuid",  
 "data": {  
 "service\_id": "uuid",  
 "version\_id": "uuid",  
 "compiler\_version": "Neuron\_v1.13.0",  
 "correlation\_id": null,  
 "source\_system": "Spark Python SDK",  
 "unique\_record\_key": null,  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "created",  
 "created\_by": "john.doe@coherent.global",  
 "created\_timestamp": "1970-12-03T04:56:12.186Z",  
 "updated\_timestamp": "1970-12-03T04:56:12.186Z",  
 "service\_uri": "my-folder/my-service[0.4.2]"  
 }  
}

[!TIP] It is recommended to close the batch pipeline once you have finished processing the data. This will help free up resources and ensure optimal performance.

## Define a client-side batch pipeline by ID

This method does not perform any action on the batch pipeline (no API call). Instead, it allows you to define a client-side reference for the batch pipeline using its unique identifier (id), which can then be used to perform various operations without having to specify it repeatedly in each method call.

### Arguments

The expected argument is the unique identifier of the batch pipeline as a string. There are no checks performed to validate the id at this stage.

pipeline = spark.batches.of('uuid')

### Returns

The method returns a batch Pipeline object that can be used to perform subsequent actions on the batch pipeline.

Some other perks of using this object, apart from the convenience of not having to specify the id repeatedly, include the ability to build statistics and insights about the batch pipeline. For instance, if you’ve built a mechanism to repeatedly pushing and pulling data, you may retrieve details such as the total number of records processed, the state of pipeline, and so on.

print(pipeline.state) # 'open' (or 'closed' or 'cancelled')  
print(pipeline.stats) # { 'chunk\_uuid\_1': 123, 'chunk\_uuid\_2': 456 }

Additionally, this Pipeline object keeps track of chunk ids. Chunk ids are important for sorting and filtering data when you have multiple chunks of data to process. This object will help autogenerate chunk ids if missing and handle duplicates to avoid collisions.

## Get the details of a batch pipeline

### Arguments

This method does not require any arguments. It will fetch the details of the batch pipeline that was previously defined using the of method.

pipeline.get\_info()

### Returns

The method returns a dictionary containing detailed information on a batch pipeline that’s been recently created.

{  
 "object": "batch",  
 "id": "uuid",  
 "data": {  
 "service\_id": "uuid",  
 "version\_id": "uudi",  
 "compiler\_version": "Neuron\_v1.13.0",  
 "correlation\_id": "uuid",  
 "source\_system": "Spark Python SDK",  
 "unique\_record\_key": null,  
 "summary": {  
 "chunks\_submitted": 123,  
 "chunks\_retried": 0,  
 "chunks\_completed": 122,  
 "chunks\_failed": 1,  
 "records\_retried": 1,  
 "input\_size\_bytes": 0,  
 "output\_size\_bytes": 0,  
 "avg\_compute\_time\_ms": 13,  
 "records\_submitted": 456,  
 "records\_failed": 0,  
 "records\_completed": 450,  
 "compute\_time\_ms": 13,  
 "batch\_time\_ms": 12234  
 },  
 "configuration": {  
 "initial\_workers": 10,  
 "chunks\_per\_request": 1,  
 "runner\_thread\_count": 1,  
 "acceptable\_error\_percentage": 0,  
 "input\_buffer\_allocated\_bytes": 70000000,  
 "output\_buffer\_allocated\_bytes": 80000000,  
 "max\_workers": 3000  
 },  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "in\_progress",  
 "created\_by": "john.doe@coherent.global",  
 "created\_timestamp": "1970-12-03T04:56:12.186Z",  
 "updated\_timestamp": "1970-12-03T04:56:12.186Z",  
 "service\_uri": "my-folder/my-service[0.4.2]"  
 }  
}

## Get the status of a batch pipeline

### Arguments

This method does not require any arguments. It will fetch the status of the batch pipeline that was previously defined using the of method.

pipeline.get\_status()

### Returns

The method returns a dictionary containing the current status of the batch pipeline and other relevant details such as the number of records processed, the time taken to process the data, and the status of the pipeline.

{  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "request\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "in\_progress",  
 "pipeline\_status": "idle",  
 "chunks\_available": 200,  
 "chunks\_submitted": 4000,  
 "record\_submitted": 1000000,  
 "chunks\_completed": 200,  
 "records\_completed": 750000,  
 "compute\_time\_ms": 82,  
 "input\_buffer\_used\_bytes": 1048576,  
 "input\_buffer\_remaining\_bytes": 68989440,  
 "output\_buffer\_used\_bytes": 402,  
 "output\_buffer\_remaining\_bytes": 68989440,  
 "workers\_in\_use": 173,  
 "records\_available": 60000  
}

Other available statuses (i.e., batch\_status) are:

* created: the batch pipeline has been created but not yet started.
* in\_progress: the batch pipeline is currently processing data.
* closed: the batch pipeline has been closed by the user.
* closed\_by\_timeout: the batch pipeline has been closed by the system due to inactivity.
* completed: the batch pipeline has completed processing all the input data.
* completed\_by\_timeout: the batch pipeline has marked as completed due to timeout.
* failed: the batch pipeline has failed to process the input data.
* cancelled: the batch pipeline has been cancelled by the user.

## Add input data to a batch pipeline

This method allows you to push input data to an existing batch pipeline. The data can be pushed in chunks, and the method will return a unique identifier for each chunk. It is also designed to facilitate the data submission in different shapes and forms.

### Arguments

The method accepts 3 mutually exclusive keyword arguments:

* inputs: a list of your records to be processed. This is convenient when you have a list of records to be processed in a single chunk. Meaning, you choose to handle the repartiotioning of the data yourself.

pipeline.push(inputs=[{ 'value': 42 }, { 'value': 43 }])

* data: an object of ChunkData type. Sometimes you may want to perform certain operations such as applying aggregations to the output data post-processing. This class lets you specify the inputs, parameters and summary separately.

from cspark.sdk import ChunkData  
  
data = ChunkData(  
 inputs=[{'value': 42}, {'value': 43}],  
 parameters={'common': 40},  
 summary={'ignore\_error': False, 'aggregation': [{'output\_name': 'total', 'operator': 'SUM'}]},  
)  
pipeline.push(data=data)

* chunks: an object of BatchChunk type. This gives you full control over the chunk creation process, allowing you not only to specify the inputs, parameters, and summary, but also to indicate the id and size.

from cspark.sdk import BatchChunk, ChunkData  
  
chunk = BatchChunk(  
 id='uuid',  
 size=2,  
 data=ChunkData(  
 inputs=[{'value': 42}, {'value': 43}],  
 parameters={'common': 40},  
 summary={'ignore\_error': False, 'aggregation': [{'output\_name': 'total', 'operator': 'SUM'}]},  
 ),  
)  
pipeline.push(chunks=[chunk])

Alternatively, you may use a helper function to create chunks and partition the data evenly across the chunks.

from cspark.sdk import create\_chunks  
  
chunks = create\_chunks(inputs=[{'value': 42}, {'value': 43}, {'value': 44}], chunk\_size=2)  
pipeline.push(chunks=chunks)

### Returns

When successful, the method returns a dictionary containing the same info as the [get\_status method](#get-the-status-of-a-batch-pipeline), but with updated values reflecting the new data that was pushed.

{  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "request\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "in\_progress",  
 "pipeline\_status": "idle",  
 "chunks\_available": 2,  
 "chunks\_submitted": 2,  
 "record\_submitted": 7,  
 "chunks\_completed": 2,  
 "records\_completed": 7,  
 "compute\_time\_ms": 8,  
 "input\_buffer\_used\_bytes": 0,  
 "input\_buffer\_remaining\_bytes": 70000000,  
 "output\_buffer\_used\_bytes": 402,  
 "output\_buffer\_remaining\_bytes": 79999598,  
 "workers\_in\_use": 1,  
 "records\_available": 7  
}

## Retrieve the output data from a batch pipeline

Once you submit the input data, the batch pipeline will automatically start processing it. Eventually, the pipeline will produce some output data, which can be pulled once available.

[!TIP] You do not have to wait for the previous chunk to be processed before submitting the next one. Spark will automatically queue the chunks and process them once the resources are available. A good practice is to monitor the status of the batch pipeline and ensure the input and output buffers are not full.

### Arguments

This method accepts an optional integer argument max\_chunks to specify the maximum number of chunks to pull. If not provided, it will pull up to 100 available chunks of output data.

pipeline.pull(max\_chunks=2)

### Returns

If there are no chunks available to pull, the method will return the status of the batch pipeline. Otherwise, it will return the output data for each chunk along with any warnings or errors that may have occurred during processing. The current status of the batch pipeline will also be included in the response.

{  
 "data": [  
 {  
 "id": "uuid",  
 "summary\_output": [[]],  
 "outputs": [{ "value": 42 }, { "value": 43 }],  
 "warnings": [null, null],  
 "errors": [null, null],  
 "process\_time": [1, 1],  
 },  
 {  
 "id": "uuid",  
 "summary\_output": [[]],  
 "outputs": [{ "value": 44 }, { "value": 45 }, { "value": 46 }],  
 "warnings": [null, null, null],  
 "errors": [null, null, null],  
 "process\_time": [1, 1, 1],  
 }  
 ],  
 "status": {  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "request\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "in\_progress",  
 "pipeline\_status": "idle",  
 "chunks\_available": 0,  
 "chunks\_submitted": 2,  
 "record\_submitted": 5,  
 "chunks\_completed": 2,  
 "records\_completed": 5,  
 "compute\_time\_ms": 8,  
 "input\_buffer\_used\_bytes": 0,  
 "input\_buffer\_remaining\_bytes": 70000000,  
 "output\_buffer\_used\_bytes": 0,  
 "output\_buffer\_remaining\_bytes": 80000000,  
 "workers\_in\_use": 0,  
 "records\_available": 0  
 }  
}

Find out more about the output data structure in the [API reference](https://docs.coherent.global/spark-apis/batch-apis#sample-response-3).

## Close a batch pipeline

Once you have finished processing all your input data, it is important to close the batch pipeline to free up resources and ensure optimal performance.

After closing a batch, pending chunks will continue to process and be available for retrieval. However, no new chunks can be submitted to a closed batch pipeline. The SDK keeps an internal state of the batch pipeline and will throw an error if you attempt to perform unsupported operation son a closed batch pipeline.

### Arguments

This method does not require any arguments. It will close the batch pipeline that was previously defined using the of method.

pipeline.dispose()

[!WARNING] Do **NOT** use the close() method to close a batch pipeline. This method is reserved for closing the HTTP client of the Pipeline API resource and should not be used to close a batch pipeline. If that happens unintentionally, you will need to start over and build a new client-side pipeline using Batches.of(id), which also means you’ll lose the internal states handled by the old Pipeline object.

Keep in mind that if the batch pipeline has been idle for longer than 30 minutes, it will automatically be closed by the system to free up resources, i.e., releasing the workers and buffers.

### Returns

{  
 "object": "batch",  
 "id": "uuid",  
 "data": {  
 "service\_id": "uuid",  
 "version\_id": "uuid",  
 "compiler\_version": "Neuron\_v1.13.0",  
 "correlation\_id": null,  
 "source\_system": "Spark Python SDK",  
 "unique\_record\_key": null,  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "closed",  
 "created\_by": "john.doe@coherent.global",  
 "created\_timestamp": "1970-12-03T04:56:12.186Z",  
 "updated\_timestamp": "1970-12-03T04:56:12.186Z",  
 "service\_uri": "my-folder/my-service[0.4.2]"  
 }  
}

## Cancel a batch pipeline

There may be occasions where you need to cancel a batch pipeline before it completes processing all the input data. It could be due to an error in the data, a change in requirements, or any other reason that requires you to stop the processing. This method allows you to cancel a batch pipeline and free up resources.

By cancelling a batch pipeline, you agree to discard all the data (pending inputs and not consumed outputs) that has been pending. The system will stop processing the data immediately and you will not be able to retrieve any output data for the cancelled chunks.

### Arguments

This method does not require any arguments. It will cancel the batch pipeline that was previously defined using the of method.

pipeline.cancel()

### Returns

Similar to the dispose method, the cancel method will return a dictionary containing the details of the batch pipeline that was cancelled.

{  
 "object": "batch",  
 "id": "uuid",  
 "data": {  
 "service\_id": "uuid",  
 "version\_id": "uuid",  
 "compiler\_version": "Neuron\_v1.13.0",  
 "correlation\_id": null,  
 "source\_system": "Spark Python SDK",  
 "unique\_record\_key": null,  
 "response\_timestamp": "1970-12-03T04:56:12.186Z",  
 "batch\_status": "cancelled",  
 "created\_by": "john.doe@coherent.global",  
 "created\_timestamp": "1970-12-03T04:56:12.186Z",  
 "updated\_timestamp": "1970-12-03T04:56:12.186Z",  
 "service\_uri": "my-folder/my-service[0.4.2]"  
 }  
}

# Workflow Example

The content above describes the building blocks of the [Batches API](https://docs.coherent.global/spark-apis/batch-apis) and its potential for efficient processing of large data volumes. How you choose to use is contingent upon specific requirements and the characteristics of the data being handled.

To further illustrate the practical implementation of the Batches API, consider the following example: the create\_and\_run script. This self-contained script serves as a demonstration of how to harmoniously use the various methods of the Batches API within a workflow. The script’s primary objective is to showcase a simple workflow that involves reading a dataset from a JSON file, pushing it to a batch pipeline, retrieving the output data from the pipeline, and displaying the pipeline’s status at different stages. The script will continue to execute until all data has been processed (unless an error occurs) and the batch pipeline is closed.

#!/usr/bin/env python  
import json  
import time  
  
import cspark.sdk as Spark  
from dotenv import load\_dotenv  
  
def create\_and\_run(batches: Spark.Batches):  
 def print\_status(status, msg):  
 """Convenience function to print the status of the batch pipeline."""  
 a, b, c = status['records\_available'], status['record\_submitted'], status['records\_completed']  
 print(f'{msg} :: {a} of {b} records submitted ({c} processed)')  
  
 # START: Main workflow  
 chunks = []  
 with open('path/to/data.json', 'r') as f:  
 data = json.load(f)  
 chunks = Spark.create\_chunks(data, chunk\_size=200)  
  
 if len(chunks) == 0:  
 print('no data to process')  
 return  
  
 batch = batches.create('my-folder/my-service')  
 print('batch created', batch.data)  
  
 if not isinstance(batch.data, dict):  
 return  
  
 pipeline = batches.of(batch.data['id'])  
 try:  
 submission = pipeline.push(chunks=chunks)  
 print('submission data', submission.data)  
 time.sleep(1)  
  
 status = pipeline.get\_status().data  
 print\_status(status, 'first status check')  
  
 while status['records\_completed'] < status['record\_submitted']:  
 status = pipeline.get\_status().data  
 print\_status(status, 'subsequent status check')  
  
 if status['records\_available'] > 0:  
 result = pipeline.pull()  
 print('result data', result.data)  
  
 time.sleep(2) # check every 2 seconds  
 except Exception as e:  
 print(e)  
 finally:  
 state = pipeline.dispose()  
 print(state.data)  
 print('done!')  
 # END: Main workflow  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # load Spark settings from .env file  
 load\_dotenv()  
  
 # create a Spark client  
 spark = Spark.Client()  
 with spark.batches as b:  
 create\_and\_run(b)

[!IMPORTANT] The script above is a sample workflow and is not intended to be used as-is in a production environment. It is meant to provide you with a starting point to build your own workflow that suits your specific requirements.

If you were to productionize this script, you would need to add graceful error handling, logging, and other features to make it more robust and reliable. You may also want to consider how you read and feed the input data to the pipeline and how to handle the output data that is returned.

Happy coding! 🚀

[Back to top](#batches-api)