**Building Spark Data Pipelines in the Cloud —What You Need to Get Started**

**Common engineering challenges and recipes for solutions**



Generated with stable diffusion by the author

**Introduction**

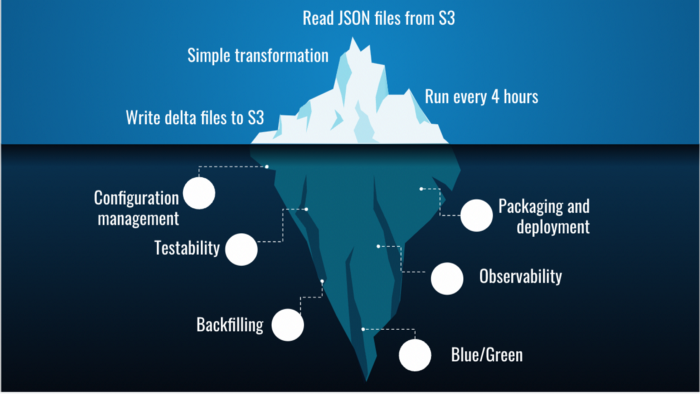
Over the last ten years or so, authoring and executing Spark jobs has become considerably simpler, mainly thanks to:

* **High level APIs** — which make it easier to express logic.
* **Managed cloud-based platforms —** highly scalable object storage and one click ephemeral clusters based on spot instances make it infinitely simpler to run jobs (and delay the need to optimize them)

While *authoring* logic in Spark and *executing* jobs has become significantly easier, *engineering* Spark-based pipelines is still a form of art with very few broadly-accepted standards and best practices.

Many teams building pipelines for the first time find that there is a steep learning curve and lack of documented best practices even for the most basic engineering concerns — such as packaging, deployment, testability, environment isolation, and observability.

Ironically, this is in sharp contrast to *SQL-based pipelines*, traditionally a black hole of engineering practices, after the community adopted DBT as a de-facto standard to address many of these same concerns.



Writing Spark code and running it on a cluster is just the “tip of the delta lake”. Image by the author

**Goal**

The goal of this post is to cover common software engineering problems and recipes for building and running **Spark-based batch pipelines in the cloud**.

This is not a complete guide — the idea is to cover some of the most common and basic areas which are applicable to virtually every pipeline.

Needless to say, the practices and suggestions described here can’t be considered “standards” — but hopefully they can provide a reasonable starting point.

The examples are based on PySpark on Databricks, but the principles should apply also to Java or Scala running on other cloud-based platforms.

**Out of scope**

* Large scale environments / complex pipelines
* Spark code or cluster level optimizations
* Analytical testing or data quality methodology

**Agenda**

**Pre-production (this post)**

**Batch data pipelines 101**

* Extract, transform, load
* Dealing with time in incremental runs
* Input and output partitioning
* Merging results

**Developing, testing and packaging pipelines**

* Basic code structure
* Tests
* Project structure
* Packaging
* Local workflow
* CI

**Environments and deployment**

* Environments
* Configuration management
* Deployment mechanism

**Production and operations (next post)**

**Running jobs in production**

* Data and pipeline observability
* Avoiding data poisoning
* Side-by-side or blue/green deployments

**Common operational scenarios**

* Retries and re-runs
* Backfilling data

Let’s go!

**Batch data pipelines 101**

**Extract, transform, load**

A batch data pipeline usually carries out one or more ETL steps.

Each step follows the pattern of:

* Extract — load data from some location (e.g. S3)
* Transform — perform aggregations, filters, apply UDFs etc.
* Load — write the output to some location (e.g. another path on S3)

Sometimes, a pipeline is modeled as a DAG of such steps.

**Incremental runs and the concept of time**

Most batch pipelines are designed to process data incrementally — i.e.:

* Read “new data”
* Process it
* Merge the output with the results from the previous runs

**Defining “new data”**

Most (but not all) batch jobs run on a schedule, e.g. daily. So most teams start off by defining “new” data as:   
“every day, run the job on the data from the day before”.

In other words, each run of the pipeline is in charge of processing a (fixed) time window of data.

This approach is known as “tumbling windows”: contiguous time frames with fixed size and no overlap between them.

**Wall time vs. event time**

A naive solution**which doesn’t work**, for processing time windows with pipelines is:

* Trigger the pipeline to run every e.g. 24 hours
* Have the pipeline code use the “***wall time***” — e.g. the value of *now() minus 24 hours* as the time window it needs to process

The reason this doesn’t work is e.g. cases when the pipeline failed over the weekend and now you need to run it on data from 72 hours ago.

*Also, most analytics is concerned with the time events happened in practice, not the time in which we received news of them happening or got round to reason about them (although in some cases, these are also important).*

So a better approach is to define the time window of each run based on the “***event time” —*** i.e. the time in which the real-world event that created the change in the data happened.

This means that each run of the job is assigned a fixed time window that is not directly dependent on the time in which the job actually ran.

(note — in some use-cases e.g. machine learning, this approach may need refinement as it can create a train-test skew).

**Using schedulers to create pipeline runs per time window**

A common approach for running a pipeline on a time window is to use a generic scheduler (e.g. Airflow, Prefect, etc.)

* The scheduler is configured to trigger the pipeline e.g. every day at some time
* Each trigger is logically associated with a distinct event-time window that needs to be processed (e.g. the previous calendar date).
* Every time the scheduler fires a pipeline run, it should pass it the trigger time
* The code uses the trigger time to figure out the time window which it needs to process.
* If our scheduler woke up later than was expected, say at 0401, it should still pass the pipeline the “right” trigger time
* If the run for a trigger failed, the pipeline should be re-run with the exact original trigger time as a parameter
* If the scheduler itself was down, it should take care to create the triggers that were missed while it was down

So there are quite a lot of subtleties and implicit contracts between scheduler and pipeline code for correctly managing time windows.

In addition, every scheduler has a slightly different interpretation of these contracts.

Here are some examples from [Airflow](https://airflow.apache.org/docs/apache-airflow/stable/templates-ref.html#variables) and [Prefect](https://docs-v1.prefect.io/api/latest/utilities/context.html#context), and [Databricks](https://docs.databricks.com/workflows/jobs/jobs.html#task-parameter-variables).

*One of the conclusions is that the pipeline needs to be able to accept a parameter representing the scheduler’s trigger time , and work out the time window based on that value.*

**Late data and watermark**

Unfortunately, data has an infuriating tendency to arrive late.

Say we want to run our batch on whole calendar days of “real world” data.  
If our scheduler fires at exactly midnight, it is possible that some of the data from the previous day is still being written, and will be missed.

A simple approach is to add a “buffer” to our scheduling time — e.g. set the trigger to fire at 0400 in order to process the previous date’s data.  
The buffer size can be guessed or even empirically determined.

A few things worth noticing:

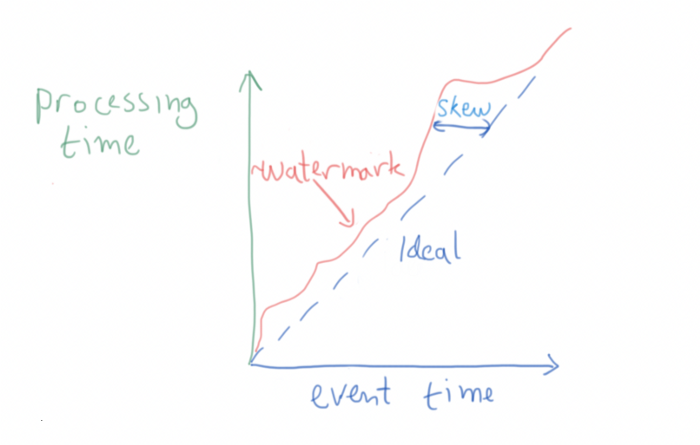
* A fixed buffer is not a guarantee for avoiding late data, it’s just a risk management technique
* There is a clear tradeoff between the size of the buffer we take (i.e. confidence of data completeness), and the freshness of the results
* If your pipeline processes the output of another batch process, you are typically more exposed to delays (e.g. when upstream batch fails)
* Data that happened to arrive later than the buffer is hard to notice unless you do something proactive about it

Turns out the subject of managing late data and deciding when it’s safe to process data with minimal risk is quite a complex issue — especially in systems where the results need to be fresh.

To tackle the tradeoff between freshness and completeness, you need to reason about the concept of ***watermarks*** — i.e. the approximate latest time we can assume is complete enough in order to process — as well as how to detect and respond to cases where older data does happen to arrive with a larger than anticipated delay.

*To summarize — for most batch jobs that run infrequently (daily/weekly), taking a buffer in the schedule is a simple heuristic for dealing with some delays in the data.*

*For other cases, you need to be giving some thought to these assumptions and perhaps reading some more on the topic, e.g.* [*here*](https://beam.apache.org/documentation/basics/#window)*.*



data watermark progress over time. Image by the author

**Partitioning the input**

When dealing with large amounts of data, we have to organize it so that it’s easy to narrow it down to the portion we want to process.

Spark supports the notion of data partitioning, where we organize the data in folder like structures on the file system, each [representing a partition of the data by some value of a column.](https://spark.apache.org/docs/2.2.2/sql-programming-guide.html#partition-discovery)

If the partitions are named according to a certain convention and contain the column name, we can add a condition about the value of the partition column, that Spark can push down and use as a filter for what data to load.

A simple example is to partition the data by date, and then add a condition that tells spark we only want to extract the data from a certain date range.

* To facilitate correct processing, especially if the pipeline is performing any aggregation, it’s best to create partitions based on the event time vs. the wall time in which they were written.
* It’s a good practice to apply this time-based filtering during the extraction phase, and leave the transformation logic neutral.
* Spark only automatically recognize a fixed set of formats for date-based partition folders.

window\_start = calculate\_window\_start(trigger)  
window\_end = calculate\_window\_end(trigger)  
df = spark.read.json("s3://my-bucket").filter(f"created\_dt >={window\_start} and created\_dt <= {window\_end}")

**Partitioning the output**

Similar to input partitioning, we usually also need to partition the output.

**When the processing is a simple transformation**

In the simplest case where the pipeline performs a transformation (vs. aggregation), it’s possible to partition the output based on the exact same column as we did for the input.

This helps us understand for each row in the input, where we expect to find it in the output.

**When the processing is an aggregation**

Say our job needs to sum the number of clicks per page in our website.

Page A had three clicks, which happened at t1, t2, and t3 within our time window.

The result for this aggregation would be a row stating that A had 3 clicks.   
This can be seen as an “aggregated” or “complex” event.  
When did this event “happen”?

One way is to define t3 as the time of the “complex event” — since this is the time which a perfect system would have emitted the value “3”.

While analytically accurate, generating per-key event time will often lead to issues managing the data during backfills and re-runs that need to overwrite results (this will be discussed later on).

An alternative that is analytically less accurate but solves many operational issues is to use a single date for the entire output — **the window end**.

While this is not the “tightest” definition of time, it is not incorrect, as it specifies a valid time in which which the system considered the event to be “true”.

*The partitioning scheme of the output impacts how you deal with backfills, as well as how you add new data increments to existing datasets. More on that in Part II of this guide.*

**Merging an incremental run’s result with the larger dataset**

When an incremental run is complete, its results need to be merged with all the previously calculated data.

An important property of the pipeline run, and the merging of output in particular, is that we want it to be *idempotent —* i.e. allow us to repeat the job run and merging several times without changing the result.

When writing the results to an object storage, we have a few common strategies :

* [Append](https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#save-modes) — add new output to the existing one, including add new files to existing partitions. Not common as re-runs for the same date range may result in data duplication
* [Overwrite](https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#save-modes) — simple strategy that works in case all runs process a fixed time windows and the output is partitioned accordingly.  
  **Overwrite only works for entire partitions and not individual records.**  
  This usually requires configuring dynamic partition overwrite (see *spark.sql.sources.partitionOverwriteMode*)
* [merge](https://docs.delta.io/latest/delta-update.html) — some of the modern file formats like delta lake or Iceberg support the notion of merging incremental results.  
  the *merge* process includes a a join-like operation between the new data to the old data by some condition.   
  Unmatched records from either side (new or old value) can be inserted or deleted. Matched records can be merged on a column-by-column basis using business logic.

Choosing the correct strategy is a delicate business.

**Merge** gives full correctness and flexibility, but is significantly more expensive in terms of computation. In the case where 99% of runs do not update the data but add to it or replace it, this is a steep price to pay.

**Overwrite** is cheap but more sensitive to edge-cases to ensure the overwrite will not lead to data loss.

**ACID properties of reading/writing partitioned data**

Imagine the following scenario: a producer is busy merging its results to the dataset, and at the exact same time, a consumer starts reading the entire dataset.

Naively, the reader may get inconsistent view of the data (with some of it written and some not).

This is a super basic concern, but up until recently requires a fair amount of careful engineering when running in the cloud since object storage solutions do not offer atomic rename operations.

If you are interested in the history of the issue, [here](https://medium.com/@vladimir.prus/reliable-and-fast-spark-tables-a6636e968f60) is some background.

Over the last 2 years or so, *this issue was more or less been solved by using a data format that supports ACID, such as delta-lake or Iceberg.*

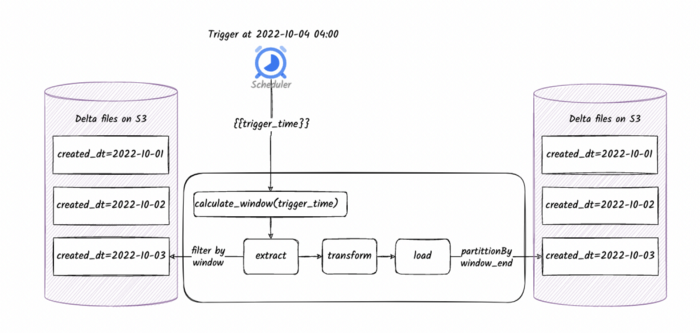
Unfortunately, not all upstream/downstream systems can read or write these data formats.

For example, if you are landing data on S3 from Kinesis firehose, or reading the pipeline’s output from some older data warehouse technology.

In these cases, you will still need to work around the issues in some other way. The [blog post mentioned earlier](https://medium.com/@vladimir.prus/reliable-and-fast-spark-tables-a6636e968f60) explains the alternatives rather thoroughly.

**Example for simple incremental pipelines**

* Partition input data by date (or hour if you plan to run on sub-daily windows).
* Control the time window size (e.g 24 hours) using configuration
* Have the scheduler trigger the pipeline with some “buffer” to allow the data to fully arrive.
* Have the scheduler pass the “trigger time” as a parameter for the run
* Read the data related to the time window using filters that spark can push down
* Partitioning the output:  
  — If your pipeline does only transformation — keep output partition identical to input partition  
  — If your pipeline does aggregation — partition the output by the end-date (or time) of the window each run handled
* Adding the incremental results :  
  — use “overwrite” mode if your data rarely updates record (but mainly adds or replaces them), ideally with an ACID-preserving data format.  
  — use “merge” mode if updates to individual records are common and you have applicative logic for applying them



A simple aggregation job running on a schedule, reading and writing partitioned data from S3. Image by the author

**Designing, testing and packaging pipelines**

**Designing the pipeline code**

**Extract and Load**

The input data is typically a part of the environment.  
Same goes to the data sink — the place where we store the output.

As a result, it’s a good idea to:

* Isolate the *Extract* and *Load* phases from the Transform phase
* Make these functions highly configurable, so that the same codebase can run on e.g. your local laptop vs. the cloud.

**Transform**

Transformations are where most of the business logic resides.

To make the transformation modular and testable, it’s common to break it down into multiple functions — e.g. have a “main” transformation function which calls into smaller functions, each taking one or more Spark Dataframes, and returning one or more Spark Dataframe.

It’s a good idea to use a functional style when writing transformation functions.

**Configuration**

Even the simplest pipeline requires some sort of support for injecting it with configuration.

Typically, you want to have the ability to configure the behavior of the extract and load so that you can run the code on different environments where data resides in different paths and formats.

In addition, there are often applicative configurations to do with the logic itself that may or may not be dependent on the environment.

More on configuration later on.

**Spark session**

Any pipeline requires an initialized Spark Session to run.

The way the session is initialized can often be specific to the environment in which the pipeline is being run.

For example some initializations that happen when running on a monitored cluster may not be applicable or appropriate to a unit test scenario.

As a result, it’s usually a good idea to initialize the session outside of the pipeline itself, and pass it to the pipeline as a parameter.

class Task:  
   
 def \_\_init\_\_(self, spark: SparkSession, config: JobConfig):  
 self.config = config  
 self.spark = spark  
   
 def main(self):  
 extracted = self.extract()  
 transformed = self.transform(\*\*extracted)  
 return self.load(transformed)  
  
 def extract(self):  
 pass  
   
 def transform(self, dataframes):  
 pass  
   
 def load(self, transformed, uri=None):  
 pass

**Tests**

If you designed the code according to the above, it should be reasonably easy to test it.

Most of the tests would cover the transformation functions, many of which take one or more dataframes and return one or more dataframes.

To set up the tests, we need to :

* Initialize the spark session for the tests (e.g. as a fixture in pytest)
* Create in-memory Dataframes and expectations
* Invoke the transformation functions
* Asserting that the output of the functions is same or similar to the expected results

See [this post](https://medium.com/@davisjustin42/writing-pyspark-unit-tests-1e0ef6187f5e) as an example.

**Unit tests**

The lowest level of unit-tests typically use a very small number of input records for each case, and these are often defined programmatically as part of the code of the test.

As an example (from the blog post above):

@pytest.fixture(scope="session")  
def stock\_data(spark):  
 schema = StructType([  
 StructField("stock", StringType()),  
 StructField("date", StringType()),  
 StructField("price", DoubleType())  
 ])  
 data = [  
 {"stock": "AA", "date": "20180902", "price": 200.0},  
 {"stock": "AA", "date": "20181002", "price": 300.50},  
 {"stock": "AA", "date": "20181102", "price": 400.50},  
 {"stock": "BB", "date": "20180902", "price": None},  
 {"stock": "BB", "date": "20181002", "price": 450.50},  
 {"stock": "BB", "date": "20181102", "price": 200.50},  
 ]  
 df = spark.createDataFrame([Row(\*\*x) for x in data], schema)  
 df.cache()  
 df.count()  
 return df

**Components or integration tests**

Here the mechanism for triggering the code is identical, but the unit under test is larger.

As a result it often requires larger input data, e.g. one which conforms to the “real” schema of the data.

It’s common to place input data for larger tests in files, organized in a data folder along with the tests themselves.

For example, to run the entire pipeline on a test input, one could write the test as follows (where the parameters are typically *fixtures*)

def test\_end\_to\_end(session, input\_path, tmp\_path, expected\_path):  
 config = {"input\_path": input\_path, "output\_path": tmp\_path}  
 job = Task(session, config)  
 job.main()  
 output = load(tmp\_path)  
 expected = load(expected\_path)  
 assert\_frame\_equals(output, expected)

**Entrypoint**

So far we’ve run parts of our logic from within a unit tests.

To run a PySpark pipeline as a standalone job (whether locally or remotely), we usually need to expose an entrypoint.

The role of the entrypoint is to:

* Parse command line parameters
* Read any configuration files if needed
* Create a session
* Instantiate the pipeline if needed
* Call the pipeline’s main function

class Task:  
 pass

def entrypoint(argv):  
 args = parse\_arguments(argv)  
 config = load\_configuration(args)  
 session = initialize\_session(config)  
 task = Task(session=session, config=conf\_model)  
 task.main()  
 print("done")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 entrypoint(sys.argv)

Note: the entrypoint is relatively generic, and can be shared across multiple pipelines (by e.g. passing the pipeline’s name as a parameter)

**End-to-end tests and running the pipeline locally**

Before launching the pipeline on remote clusters, its usually very useful to be able to run it end-to-end locally, with an in-memory runner.

Here, we use the entrypoint to do all the initialisation of the session and configuration objects, controlling e.g. the input/output paths for the job.

Writing tests that run the pipeline end-to-end is possible since the entrypoint allows you to inject command line arguments.

Debugging is possible by creating a launcher from within the IDE that invokes the entrypoint’s module with the right parameters.

Finally, you may invoke the pipeline from a shell, either before or after packaging it.

**Project structure and packaging**

So far, all we have is a pretty standard Python project.

We will need all the standard scaffoldings —

* Requirement management
* Project packaging tool
* github actions
* pre-commit hooks
* etc.

You can adopt your favorite modern Python package template such as [this one.](https://github.com/fedejaure/cookiecutter-modern-pypackage)

**Packaging**

What we have so far is a pure Python project, so it makes sense to package it as a .whl file using a standard packaging mechanism based on setuptools.

if you followed a modern Python project template, you will have selected your packaging mechanism (setup.py, pyproject.toml, poetry-style etc.)

It’s often a good idea to expose the entrypoint function as an API for the package, e.g. [like so](https://setuptools.pypa.io/en/latest/userguide/entry_point.html#console-scripts).

This enables you to install the packaged pipeline, and trigger it directly:

pip install my-pipeline  
my\_task arg1 ...

In most cases, at least in Python, there’s not much mystery about packaging, so you should be able to package your local code on your laptop without much trouble, which is useful for local testing too.

**Optional — running local code on a remote cluster**

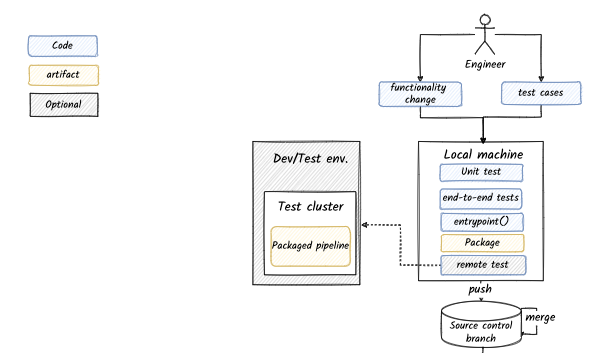
For testing purposes, it’s often useful to be able to deploy your local code (after packaging) to a remote cluster in a dev/test environment.

This can save a lot of time in finding and fixing trivial environment related issues (vs. waiting for some CI/CD pipeline to fail later).

If your deployment mechanism is well designed, this is usually very feasible.

We’ll get to discuss deployment later on in this post.

**Local workflow summary**

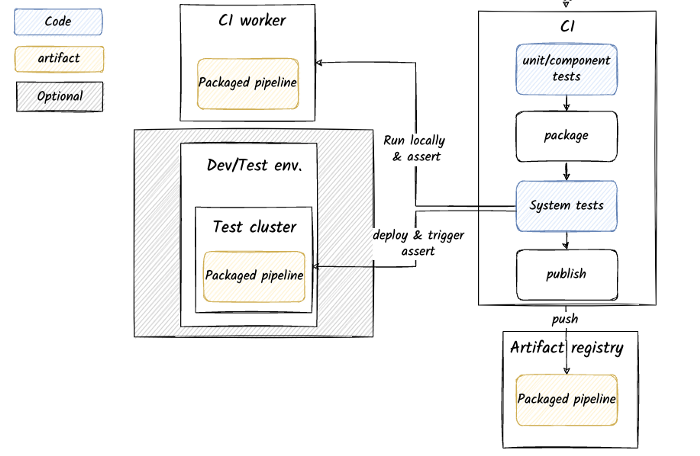


Local development workflow — image by the author

**CI**

Here you can follow a classic Python CI workflow using e.g. github actions, that includes:

* Check out the code
* Run pre-commit hook for linting and other enforcements
* Run unit tests
* Run larger tests (e.g. end-to-end test)
* Package the pipeline
* Push it to some (private) artifact registry



CI for testing, packaging and releasing data pipelines. Image by the author

We’ve touched upon the subject of deployment to a remote cluster, now it’s time to tackle this topic more in depth.

**Deployment and environment management**

Here is where things start to get tricky.

* What does an “environment” mean in a data pipeline context?
* What does an environment include?
* How are environments isolated from each other?
* How do the different environments influence my codebase and dev. process?
* What artifacts to I need to deploy each time?

Let’s start unpacking these questions.

**What are environments and which ones do I need?**

Generally, an “environment” is the set of resources in which the software you are building can be deployed and run under various scenarios.

It’s common for software development processes to include some or all of the following environments:

* **Local** — running the system on a local machine or CI worker (see local workflow above).
* **Dev/Test** — this environment may be personal, allocated per team, or shared across multiple teams. Typically used for early testing of features and optionally CI
* **QA** — once an official version or release candidate was built shipped, some teams deploy it to a separate, clean environment for QA (mainly when QA is partially manual)
* **Staging** — prior to releasing the software to production, we may “stage” it in a production-like environment and observe it’s behavior under production-like conditions
* **Load-test** — when you need a dedicated set of resources to generate and test the system under load
* **Production** — where the system can run and impact the end user

**What does an “environment” for Spark pipelines include?**

At the very minimum, the environment includes :

* Spark clusters (or the ability to spin them up)
* Input data (or tools that can generate the data)
* Sinks for writing pipeline output data (e.g. S3 bucket or databases)

In addition, the environment *may* include:

* Other pipelines (e.g. upstream and downstream ones)
* Data catalogs (e.g. hive metastore)
* Schedulers/orchestrator tools
* Monitoring tools
* More

**Reasonable starting point (example for Databricks on AWS)**

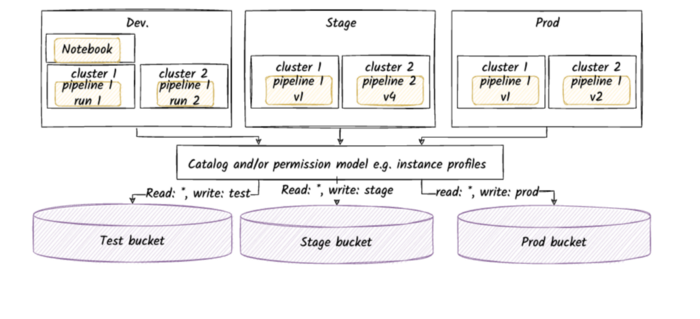
* **Local** — in-memory Spark (reading from local FS or S3)
* **Dev/test, Stage, Prod** — separate Databricks [*workspaces*](https://docs.databricks.com/administration-guide/account-settings-e2/workspaces.html), sharing the same Databricks and AWS account.
* **Storage** — separate buckets for each workspace.   
  All environments may have read-only permissions to each other’s buckets
* **Monitoring** — same account for staging and production with different labels on the metrics and ability to filter by environment in the dashboards.

To scale to more teams, you may decide to open separate dev/test workspaces per team.

**Environment level isolation is not enough**

There are often much fewer environments than pipeline runs using them.

For example, we may be working on two different pipelines in dev/test, or running two versions of the same pipeline in production side-by-side.



Multiple job runs share the same environment. Image by the author

It follows that much of the isolation between pipeline runs needs to come from the way we deploy the pipelines into the environment.

**Deployment**

**What does a deployment include**

A *deployment* of a pipeline to an environment usually includes:

* **Name** for the for the deployed pipeline
* **Pointer to the binary** we need to run (e.g. Python packages in private PyPi repo)
* **The command** for running the binary
* **Applicative configuration** (e.g. command line arguments)
* **Cluster configuration** — including machines, roles, spark configurations, init scripts, env. variables etc.
* **Scheduler configuration**
* **Monitoring and alerting configuration** when relevant

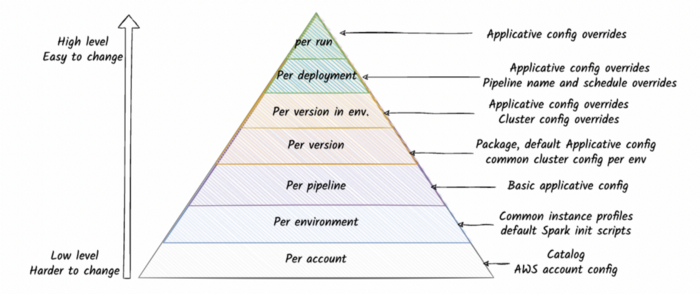
**Deployment as config**

Most of these deployment options are highly structured.

Most cloud Spark platforms have [structured APIs for deployment](https://docs.databricks.com/dev-tools/api/latest/jobs.html) that accept some variation on the above information as JSON payloads.

**Hierarchical configuration management**

To avoid repeating the configuration each time we deploy a pipeline, we may want to manage the configuration hierarchically.



Hierarchy of configuration. Image by the author

The lower level of configurations are part of the account or environment and are placed there when these are first set up (and rarely change).

*The rest of the configuration options that are pipeline specific need to be managed along with the codebase of the pipeline since they may change as part of the process of developing the logic.*

**Applicative configuration**

Virtually every pipeline needs to expose some way to configure its behavior externally.

For example:

* Controlling the input and output paths on S3
* Setting a limit on a query in a test environment
* Setting the size of the time window on which to perform aggregations
* etc.

If the number of parameters you want to control is significant, it makes sense to organize them into configuration files.

Following the hierarchical configuration concept, we may want to have a configuration mechanism that allows:

* Specialization of configurations managed in files (e.g. per environment)
* Overrides of values via the command line arguments

A great tool that enables all of this and more is [hydra.cc](http://hydra.cc)

my\_pipeline -c /path/to/hydra/configuration/root +env=test ++job\_config.some\_key="x"

A guide to hydra is beyond the scope of this blog post.

**Deployment mechanics**

The deployment procedure needs to perform the following actions:

* **Build the package** (optional)
* **Deploy the package** to an accessible location for the cluster e.g. PyPi repo (optional)
* **Deploy of applicative configuration files** to an accessible location for the cluster (e.g. shared FS available to the Spark driver)
* **“Compile” the runtime configuration** for the specific deployment of this pipeline — figure out the cluster, the name of the pipeline, the right command to invoke the pipeline, scheduler configuration, monitoring configuration etc.
* **Call the API’s to create the cluster and register the pipeline** correctly in the environment, with all of the above information.

A great tool for doing all of the above on Databricks is [dbx](https://dbx.readthedocs.io/en/latest/).

* The deployment is invoked from the command line.

dbx deploy --environment test --workflow my\_pipeline --config ....

* It comes with a .yaml file that controls the build command, as well as most of the other aspects of the runtime configuration.
* Generally it organizes the deployment by environment and pipeline.
* Much of the applicative configuration per deployment is overridable from the deploy command.
* You can add some more dynamic behavior e.g. override the name of the Databricks job created, using Jinja templating and environment variables.

**Project structure with dbx and hydra-based configuration**

.  
├── .github  
│ └── workflows  
│ ├──ondeploy.yaml  
| └──onpush.yaml  
├── .dbx  
│ └──project.json  
├── conf  
│ ├── deployment.yaml #dbx  
│ └── tasks  
│ └── my\_task #hydra  
│ ├── my\_task.yaml   
| ├── test  
│ ├── stage  
| └── prod  
├── my\_pipeline  
│ ├── entrypoint.py  
│ └── tasks  
│ └── my\_task.py  
├── tests  
│ ├── data  
│ ├── unit  
│ │ └── tasks  
│ │ └── my\_task\_test.py  
│ └── system  
│ └── system\_test.py  
├── .pre-commit-config.yaml  
├── pyproject.toml  
├── setup.cfg  
├── setup.py  
└── README.md

**Summary**

In this post we covered the basic pre-production concerns for Spark-based data pipelines in the cloud.

None of the above information can be considered novel per se, but I hope it can save the reader some effort in both identifying and / or designing solutions to these common problems.

The next installment will focus on running and operating the pipelines in production.

Thanks to Ohad Raviv and Ben Huberman