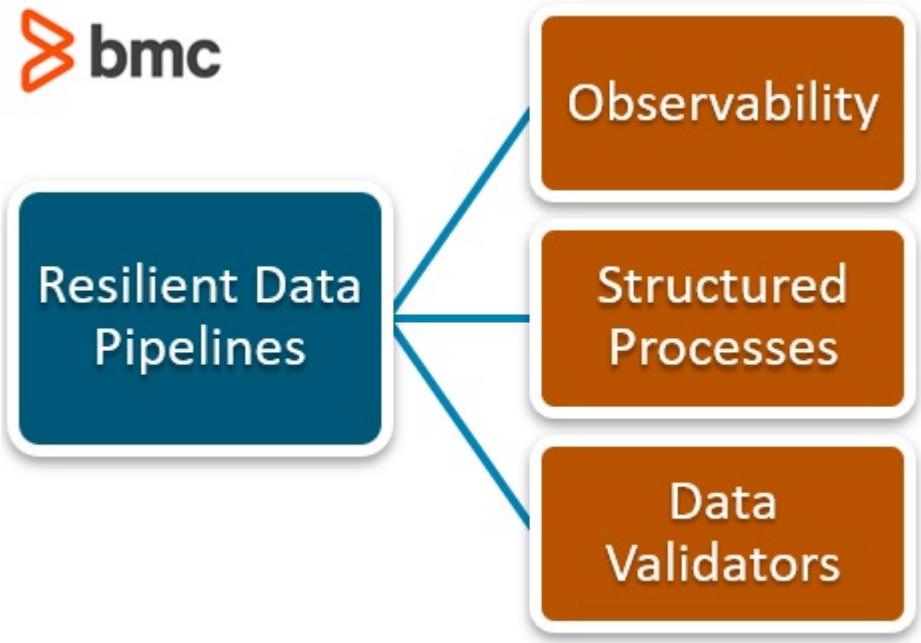
3 KEYS TO BUILDING RESILIENT DATA PIPELINES

Resiliency means having an ability to adapt. Resilient data pipelines adapt in the event of failure. Data pipelines are meant to transport and transform data from one point to another. So, a resilient data pipeline needs to:

1. Detect failures
2. Recover from failures
3. Return accurate data to the consumer



To achieve this, you'll need to form a resilient data pipeline, which requires three components:

- Observability
- Structured processes, including idempotency and immutability
- Data validators

Observability

Errors need to be seen to be corrected quickly. If the error goes unseen, it cannot be fixed. Good logging practices help surface errors, and well-written logs allow the error to be identified and located quickly. Developers, usually the people who will be reading these logs, need to know what has happened in the event of an error.

Logs should communicate what has happened, and what action needs to be taken to not encounter the failure. There are three things a good log message contains:

1. Logs should use an appropriate, and consistent, category for what error has occurred.
2. Logs should specify what exactly has happened to cause the error.
3. ogs should inform the user what action needs to be taken to correct the error.

If the log can give the developer action towards fixing the issue without having to probe the code for the problem, the log is a good log.

Logs can also be written at both the pipeline and the task levels. Logs should answer:

- Why did the pipeline task fail?
- When was the pipeline scheduled?
- Whether you are using your own processors or renting from a third-party, each processor should produce logs and you should get access to the third-party's or write in your own logs to widen your own visibility window.

Finally, after data has gone through the data pipeline, logs should be available to report the data has successfully traveled through the pipeline.

Observability is also possible by examining the metrics of your pipeline resources. Check out the latency of your system, the queued time for batches, and how much resources your batches consume during each run.

Structured Processes

Resiliency improves when the infrastructure of the data pipeline uses better structures. The aim in the system's design is to use processes in the transport of the data that give an expected and predictable result. To further qualify, the data should arrive as expected even in the event of failure.

Idempotence and immutability are processes that help return data in the event a processor is unavailable, stops half-way through the shipment, or gets triggered multiple times. They help accomplish two things:

- They ensure good data gets to the end user.
- They ensure data arrives in the event of a systems failure.

Idempotence

Idempotence is a function that will return the same result on repeated instances of execution. Functions like absolute functions and rounding functions are idempotent. Not all functions are idempotent, but functions can be made to be idempotent.

Deletion functions can be written as idempotent. In a list of data , if the goal were to remove the number 2 at index 2, the function to delete the second index in the list is not idempotent, because if the function runs multiple times on the list, it continuously removes whatever number happens to be at the second index. In this example, the function would remove the number 2, then 3, then 4, and so on…

One way the function can be rewritten as idempotent is by saying if the number in the second index is a 2, delete it. This function can be run over and over and gets the desired output of deleting the number 2 from the list. Idempotent functions can vary depending on the dataset and what the result is intended to be. (Wikipedia lists some more examples of idempotent functions.)

Immutability

An immutable infrastructure decreases complexity and allows for simple reasoning about the infrastructure.

Traditionally, mutable servers could be updated and transformed while they were being used. This allowed developers to update any software or transform data on the server while data wrote to it. The problem with this is that each update tweaks the server a little and each server is just a little bit different than the other, with different version numbers—1.34, or 1.74, or 1.22, or whichever.

For resiliency, trying to identify the error in one of these systems can get to be very complicated. Versioning the pipeline is also very difficult when dealing with a morphing mixture of server versions. Similarly, trying to update the entire infrastructure from version 1 to version 2 encounters its own challenges as each server gets updated from 1.34 to 2.0 or 1.74 to 2.0.

Immutable means that data cannot be deleted or altered, "mutated", once written on the server. Immutability, like idempotence, is a good design practice what for resilient infrastructure because it

allows simple versioning of data pipelines, and, when errors occur when building servers, these errors are separated from actual tasks being processed.

In the immutable infrastructure, Step 1 may be to create a VM instance and Step 2 may be to write user data to the instance. Failure can occur in any number of ways when creating the VM instance, and the immutable infrastructure can run and rerun step one until a stable instance is ready, then start adding data. When the infrastructure was mutable, failure is more common, complexity is greater, versioning is more difficult.

Data Validators

Data validation occurs upon the arrival of the data to the consumer. Data validation is meant to quickly process the data from the pipeline and verify that it is the correct data the consumer is looking for.

Data validators will check for:

1. **Data types**, i.e., Integer, String, Booleans, etc.
2. **Constraints**, i.e., email is a university email, phone number has 10 digits, value is greater than zero.
3. **Quantitative validation** usually requires domain-specific knowledge. Quantitative validation would be things like the height of an NBA player is likely not four-feet tall. Or the dinosaur fossil is not likely three weeks old.

Validators are good to ensure that the data that's arrived to the consumer is appropriate, but they can run into a few errors. Depending on how the validation takes place, it can increase pipeline latency, and, of course, there could just exist bad code and the validator doesn't fully do what it is supposed to.

Adapting when failure occurs

Resilience is adapting in the event of failure. By designing observability into the pipeline, failures are announced and seen and allow action to be taken.

We have seen through different structures, like idempotence and immutability, the infrastructure of the pipeline can be built to be more resilient. And, because we want to make sure the data gets to the end appropriately, we place data validators on the pipeline that ensure the appropriate data is being returned to the consumer. With proper logs in place throughout the pipeline, failures will be exposed and, whether through machine or human correction, the system is free to adapt.

Additional resources

For more on this topic, browse the BMC DevOps Blog or see these articles:

- Resilience Engineering: An Introduction
- How to Create a Machine Learning Pipeline
- Simplifying and Scaling Data Pipelines in the Cloud
- Machine Learning with TensorFlow & Keras, a multi-part tutorial