DataCater

White Paper
December, 2021
Author: Hakan Lofcali

Cloud-Native Data Pipelines

Accelerating data development by adopting cloud-native principles.

Cloud-Native Data Pipelines

In the last decade, the adoption of cloud computing grew exponentially. The shift from monolithic applications to microservices to cloud-native software has changed the responsibilities and capabilities in application development.

The development and the operation of software were treated as two different crafts. With the rise of cloud technologies, such as Kubernetes, the software industry merged these skill sets into one, allowing one role, "DevOps", to take care of both tasks. DevOps tools were created to meet the new requirements of a "You built it, you run it" culture. Principles, which were at the heart of this successful shift, include containers as deployment units, declarative runtime descriptions, continuous integration and deployment, observability, and elastic scalability.

Cloud-Native

https://www.cncf.io/about/who-we-are/

"Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."

Data workers were largely untouched by this shift, and we still see clear separation in development and the operation of data-driven applications.

In this whitepaper, we outline the advantages of adopting cloud-native computing principles to the building of data pipelines and the workflow of data engineers, data scientists, and ML engineers. We aim to enable data pipeline developers to operate their data pipelines efficiently and effectively. For this, we outline how cloud-native principles apply to data workflows and where we see shortcomings in the current landscape of tools.

We present the benefits of containers for running data pipelines, demonstrate the improved transparency that can be unlocked by the "Sidecar" pattern, and discuss means to unlock declarative descriptions of data pipelines. The closing chapters on "DataOps" and "Scalability" summarize the impact of cloud-native principles and tooling on data workflows and how data processing must adapt to these to leverage the full potential of the cloud-native movement.

Runtime Consistency with Containers

In cloud-native environments, applications are deployed as containers. In "Principles of Container-Based Application Design"[1], RedHat defined a set of principles, which are nowadays industry standards for designing containerized applications. This section carries the foundational work of RedHat over to building and deploying data pipelines in the cloud era:

Image Immutability Principle (IIP): "..., and once [Containers are] built are not expected to change between different environments". For data pipelines, this should even be true for the same environment and different revisions of data pipelines. Data is at the heart of every organization, and the processing of data needs to be auditable in hindsight as well as runtime. IIP gives data teams and their organizations the ability to pinpoint the code that was used to transform and gain value from their data.

Self-containment Principle (S-CP): "This principle dictates that a container should contain everything it requires at build time." Relying only on the availability of the Linux kernel effectively allows any library or even language to be used at data pipeline definition/creation time. Today, the majority of data transformations and data science work is done in Python with an abundance of choice when it comes to feature engineering, data transformation, and ML/statistical model training. S-CP allows data teams to use any of their favorite tools and libraries to effectively build a runtime environment tailored to their needs.

Runtime Confinement Principle (RCP): "This RCP principle suggests that every container declares its resource requirements [CPU, Memory, Storage] and pass that information to the platform." Data pipelines can and should declare their resource requests. Having the ability to change these at runtime makes for great adaptability and leads to less downtime or crashes of data pipelines. This principle improves the transparency for analyzing resources used by different data teams across organizations. RCP allows for isolation of resource consumption and therefore failures, which are caused by "rogue" data pipelines consuming more resources than expected.

"Image Immutability", "Self-containment", and "Runtime Confinement" are three container based application principles, that greatly benefit developing and deploying data pipelines.

1 https://www.redhat.com/en/resources/cloud-native-container-design-whitepaper

"Image Immutability", "Self-containment", and "Runtime Confinement" are three of RedHat's seven principles. These three principles have a high impact on developing, deploying and operating data pipelines. Applying these to data pipeline runtimes will result in more consistent testing, higher transparency of deployed code, and better over-all observability.

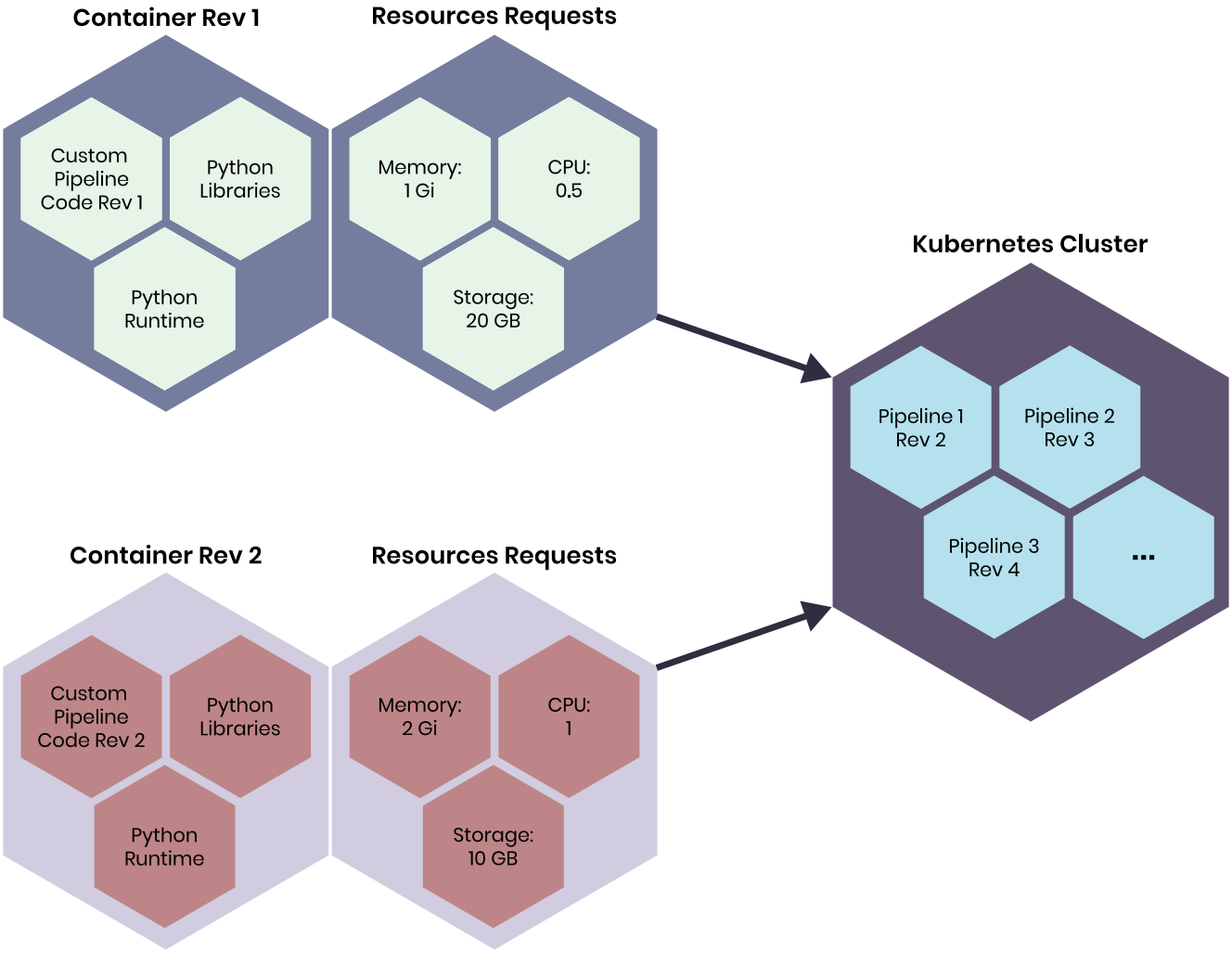

Figure 1: Container Based Data Pipelines

Sidecars: Non-intrusive Observability Pattern

Containers and Kubernetes unlock the potential for real-time and non-intrusive monitoring, logging, and debugging features. A common pattern for this is the use of sidecar containers. Sidecars are automatically injected into a Kubernetes Pod and resemble the standard practice to increase observability, control, and auditing potentials for cloud-native workloads.

Injecting sidecars to, for instance, forward network connection and application logs of data pipelines allows for centralized observation. Utilizing open-source tools for service meshes (e.g., OSM, Istio, Linkerd, etc.) and alerting & monitoring (e.g., Prometheus, Grafana, ELK, etc.) allows for maximum transparency and control integrated into established technologies.

Cloud-native tools unlock these capabilities without putting a burden on the developer of data pipelines, since these benefits are automatically provided by the underlying platform at deployment time without the container being aware of them.

Cloud-native platforms for data pipelines might inject sidecars into workloads, so that organizations can have full transparency on the “Who?”, “What?”, and “Why?” for all data that is being processed, transformed, and loaded into analytics databases or ML models (referred to as “Data Sinks” below).

Sidecar containers allow for so-called circuit breakers. Circuit breakers are processes, which isolate a given workflow from the rest of your applications and databases. Detecting malicious or unexpected behavior of a data pipeline is enabled by the collected data. Once, a data pipeline is classified as a threat, circuit breakers can isolate the given process to reduce harm to the software system as a whole.

Cloud-native tools unlock observability, without placing a burden on data developers. Sidecars are injected by the underlying platform.

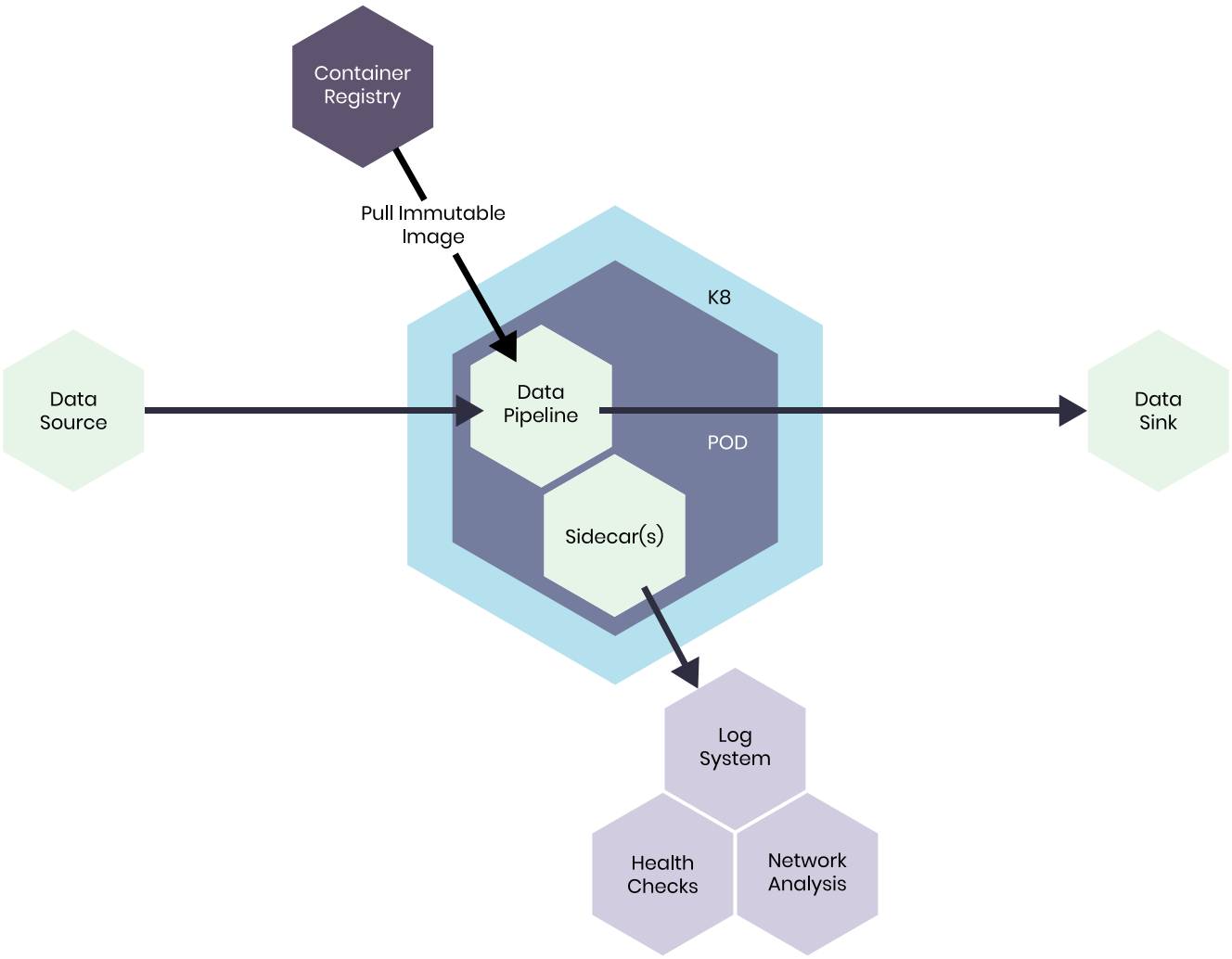

Figure 2: Observability for Data Pipelines

Declarative Data Pipelines for Operations

As stated in CNCF's definition of cloud-native technologies, declarative APIs are a cornerstone of cloud-native technology. Among APIs, we also have tools like Hashicorp Terraform, AWS CloudFormation, Kubernetes Manifests, and others that commoditize "Infrastructure as Code" and enable developers to manage their infrastructure in a declarative manner.

A declarative approach allows for more reliable, resilient, and reproducible deployments and faster iterations in development. To allow the declarative definition of data pipelines, the underlying platform needs to fulfill a couple of prerequisites.

A common pattern in cloud-native environments is the outsourcing of the application state, e.g. moving session affinity to load balancers, implementing cache management with key-value stores, etc. State-managing technologies are consumed as services, which are declared as dependencies and either configured during runtime or at deployment time.

To enable declarative descriptions of data pipelines, we require services to keep some state and connect individual data pipelines to these. Each data pipeline requires the ability to outsource:

1. Which data has already been processed?
2. How to connect to a given data source or sink?
3. How many resources can or shall be used?

Questions 2 and 3 are easily answered by an orchestration system like Kubernetes. Connections to data sources and sinks can be defined through a combination of *ConfigMaps* and *Secrets*, which can be referenced at container startup time. New data pipelines might be configured to use a certain *Secret* managed by the operator deploying the pipeline.

Container orchestration systems provide means to declare how many resources a given application requires. Such *ResourceRequests* can be used to schedule a data pipeline.

Unlock declarative data pipelines by outsourcing state management to an event streaming system like Apache Kafka.

Which data has already been processed? DataCater uses Apache Kafka to keep track of records processed by committing offsets. Event-sourcing systems source events from data stores and can retain the data for a requested period of time in their local storage. Sourced events can be emitted or polled by data pipelines to apply transformations and load the processed data into target data sinks.

Therefore, an event-sourcing system such as Apache Kafka Connect is crucial for modern cloud-native data pipelines. Having a service tracking what data has processed by which consumer unlocks the potential for having a declarative approach to data pipelines.

With state management provided by a platform with the above components, we can define Data Pipelines as a set of three components: data sources, *filters and transformations, and data sinks*. The following *Figure 3: Data Pipelines as Code* illustrates a cloud-native data pipeline.

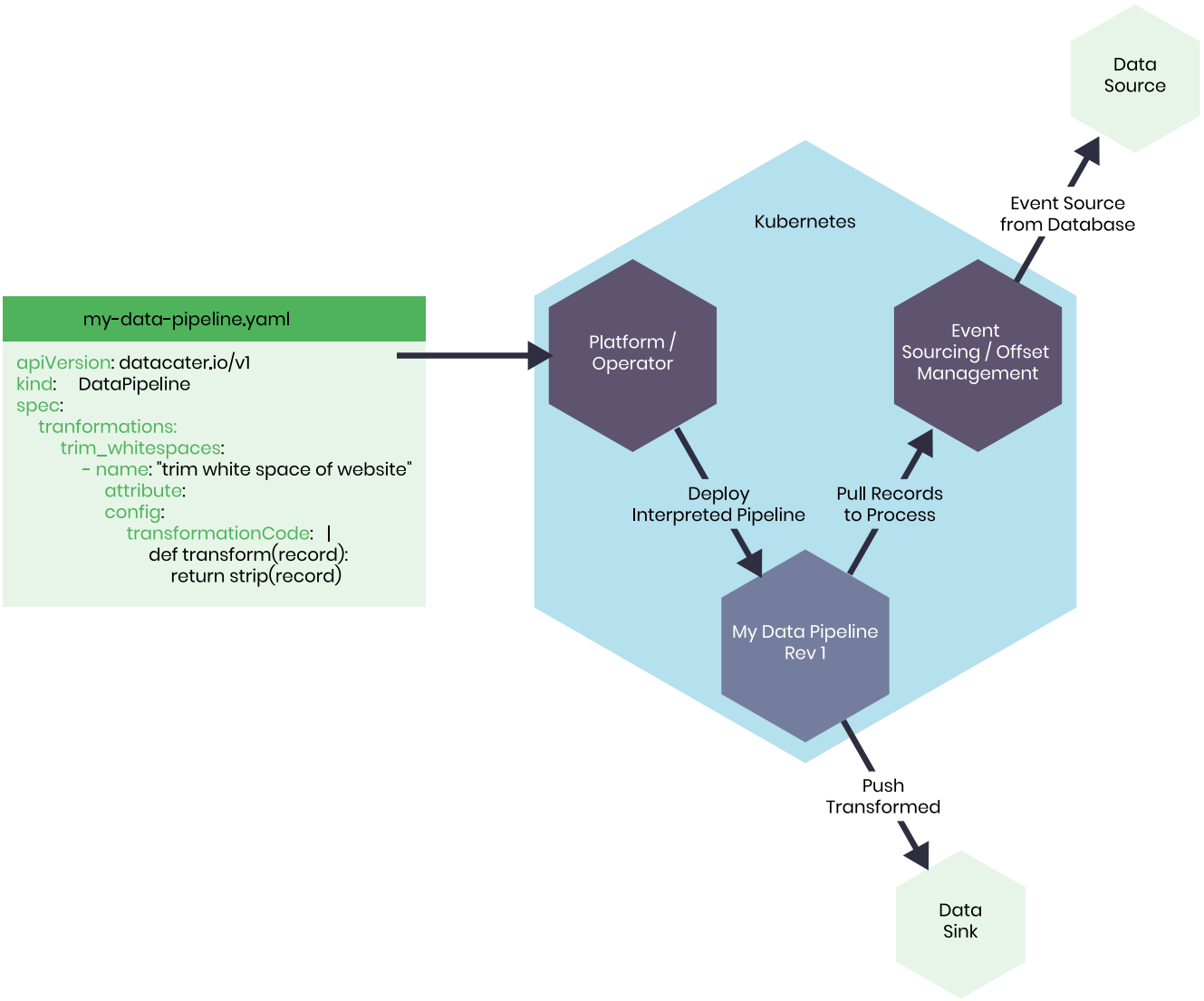

Figure 3: Data Pipelines as Code

DevOps and DataOps

In the scope of this whitepaper, we define DevOps as "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality".[2]

The ultimate goal of DataOps would be to reduce the time needed for developing and deploying data pipelines. Breaking down the parts of the above definition, we can derive tools required to implement automated deployments to production upon code changes in data pipelines.

The status quo of developing and operating data pipelines unveils a couple of conceptual flaws when it comes to automated roll-outs of new pipeline revisions. We addressed two needs of data pipelines, runtime consistency and declarative definition, in former chapters. This chapter focuses on automatically ensuring the quality of changes applied to data pipelines.

Application development in a DevOps fashion ensures high quality by having consistent builds and fully-automated unit, integration, and end-to-end testing of new revisions. Data pipeline developers do not have access to similar tooling and cannot efficiently test changes before rolling them out to production. Reasons for this include:

1. Runtimes are not isolated. Resource sharing makes it almost impossible to replicate behavior from testing environments to production environments, and vice versa. Production environments typically run much more workloads in parallel and are equipped differently, making them behave differently.

2. Frameworks like Apache Spark, Apache Hadoop, etc. require a running (simulated) cluster for their programmatic APIs to work. As a consequence, unit tests either become long-running procedures or developers test their data transformations separately and "hope" that the corresponding frameworks APIs behave the same way.

Reduce data pipeline development iterations by switching to an event streaming execution model.

2 Bass, Len; Weber, Ingo; Zhu, Liming (2015). *DevOps: A Software Architect's Perspective*.

We propose a different approach to processing data. From our experience, most data sets are in motion and change over time. Instead of assuming that a data set can be processed in one go as a batch, we prefer considering data sets as continuous flows of information. As a consequence, we never assume that a data set is complete but always expect that - at a certain time - we can only see a window of an endless stream of data. In other, slightly more technical terms, we suggest perceiving batch jobs as a subset of event streaming where batches might resemble a window, defined by time range or amount of records, of data change event streams.

Moving from batch processing to streaming unlocks automated quality assurance for data pipelines. Container-based data pipelines are self-contained and define their needed resources at deployment time. Self-containment isolates data pipelines from each other and allows an orchestrator, like Kubernetes, to ensure the availability of compute resources before starting the transformation of data.

Windowed streams can be emulated by the same principles as applied in continuous integration pipelines. For efficient testing, you might reduce the size of the window, if your data pipeline requires aggregates. However, there is no need to change the actual code to adhere to a certain specialized test runtime, as the code you test with is the exact same code you run in production.

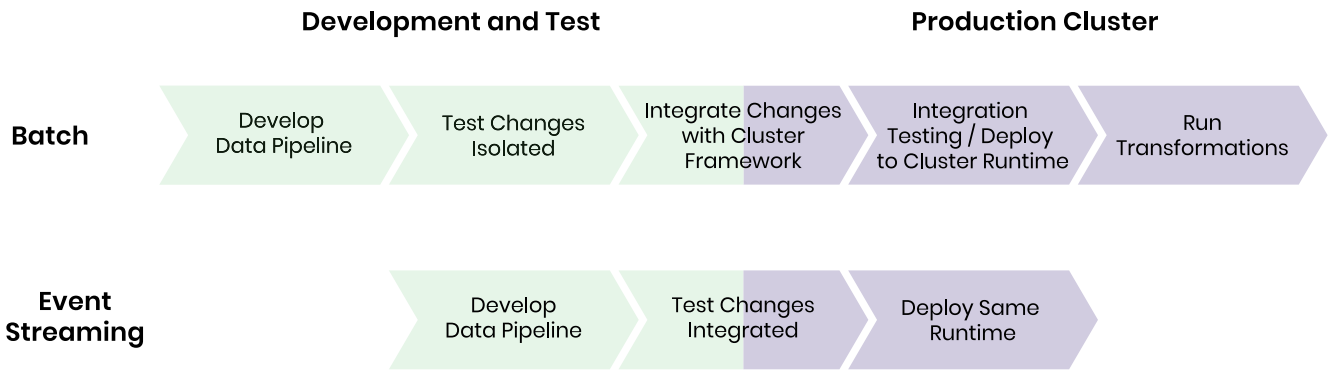

Figure 4: Development Loop for Batch and Streaming

Scalability - changing data processing

Gartner defines scalability as follows:

> *"Scalability is the measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands."*[3]

One major reason why cloud providers are tremendously successful is their "pay as you go model", which requires scalability built into applications. Today's ETL, ELT, and data pipeline frameworks work by loading large batches of data. Hence, the amount of resources requested, especially CPU, disk storage, and memory, are calculated against the peak load. This in it itself is not a scalable approach, as we cannot adapt over time.

Kubernetes and other cloud-native technologies give us the ability to elastically scale our resource needs at runtime: Requesting more or less resources is a common approach. With event streaming, we can make use of the scalability capabilities of modern cloud-native runtimes.

Once we make the shift to event streaming, the need to calculate peak loads becomes obsolete. Container-based data pipelines can automatically calculate their resource requirements as a function of window size. In its most basic version, the function would look like:

> *R(window_size) := window_size * record_size + constant*

With event streaming as the standard mechanism to extract, move, transform, and load data to databases. We can leverage scalability of underlying cloud native platform resource allocation to optimize our usage of resources. Kubernetes offers so-called *Horizontal Pod Scalers (HPA)*. HPA's are automatically managed by Kubernetes for certain application types like *Deployments*. As we outsourced tracking state, we can deploy streaming data pipelines as stateless applications and Kubernetes takes care of varying the resource usage. This is depicted in the following figure.

Adopting scalability to data pipelines by specializing on event streaming and leaving batch models behind.

3 https://www.gartner.com/en/information-technology/glossary/scalability

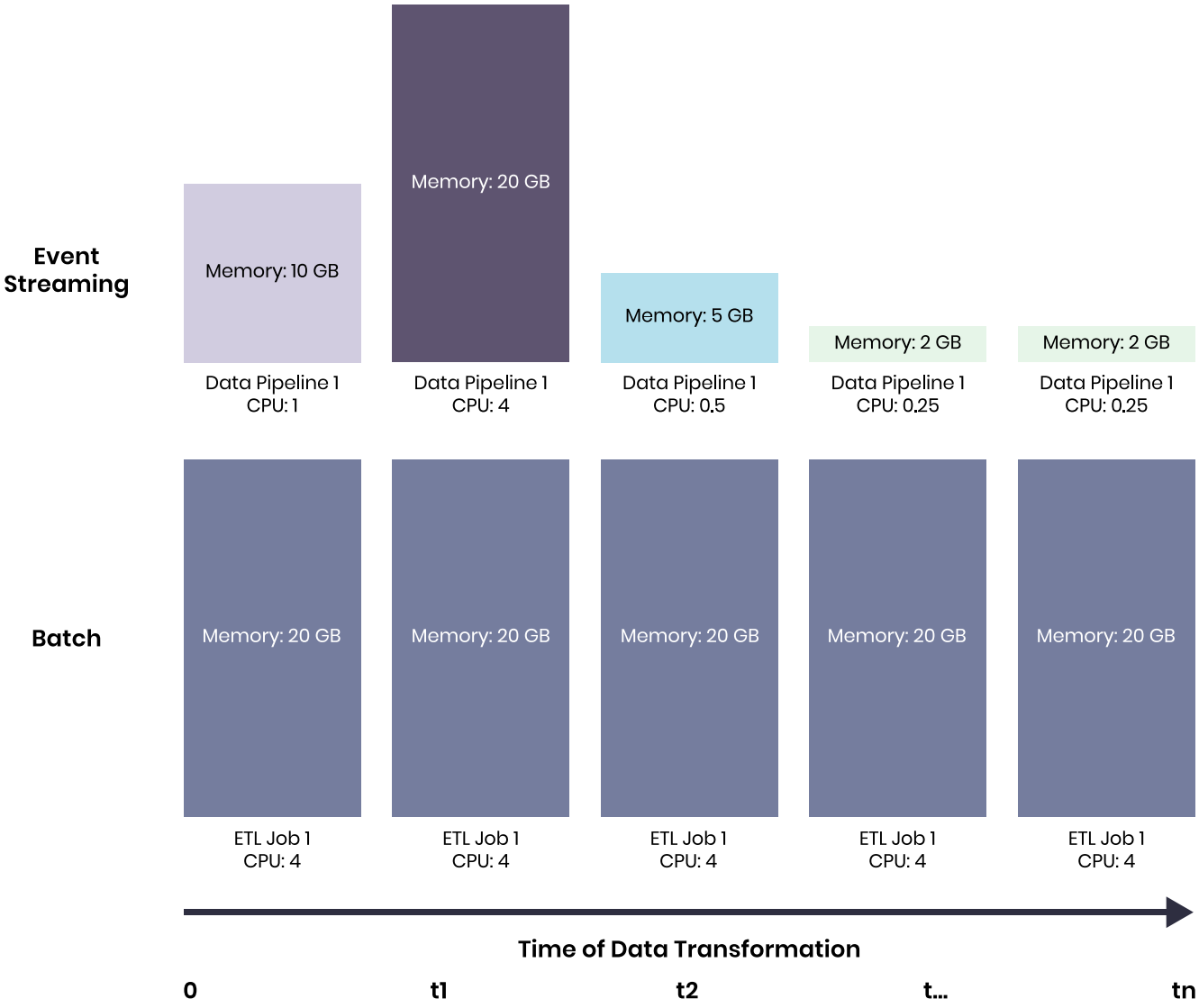

Figure 5: Scalability for Batch and Streaming

Conclusions

Cloud-native environments change the way we, as an industry, should work with data. This white paper walked through the advantages of applying cloud-native best practices to data workflows like ETL, ELT, and data pipelines in general.

We discovered shortcomings in tooling and platforms, which need to be addressed to effectively transfer learnings from DevOps to data workflows. DataOps can be achieved, if we advance the development of data pipelines as follows:

- Reduce steps for continuous integration by using the same runtime for development, testing, and production is enabled by container images as immutable contexts.
- Prefer (windowed) streams over batches for a higher elasticity instead of blocking resources for peak loads.
- Use modern message brokers, such as Apache Kafka, for managing the state of data pipelines.

Want to learn more?

Reach out to us!

info@datacater.io

DataCater

Cloud-Native Data Pipelines

December, 2021

Want to learn more? Reach out to us!
info@datacater.io