O'REILLY®

Compliments of Lightbend

Designing Fast Data Application Architectures

Gerard Maas, Stavros Kontopoulos
& Sean Glover

Designing Fast Data Application Architectures

Gerard Maas, Stavros Kontopoulos, and Sean Glover

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Designing Fast Data Application Architectures
by Gerard Maas, Stavros Kontopoulos, and Sean Glover

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 9547.

Editors: Susan Conant and Jeff Bleiel
Production Editor: Nicholas Adams
Copyeditor: Sharon Wilkey
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

April 2018: First Edition

Revision History for the First Edition
2018-03-30: First Release

This work is part of a collaboration between O'Reilly and Lightbend. See our statement of editorial independence.

978-1-492-04487-1

[LSI]

Table of Contents

Introduction v

1. The Anatomy of Fast Data Applications 1
A Basic Application Model 1
Streaming Data Sources 2
Processing Engines 3
Data Sinks 5

2. Dissecting the SMACK Stack 7
The SMACK Stack 7
Functional Composition of the SMACK Stack 8

3. The Message Backbone 11
Understanding Your Messaging Requirements 12
Data Ingestion 12
Fast Data, Low Latency 14
Message Delivery Semantics 15
Distributing Messages 15

4. Compute Engines 17
Micro-Batch Processing 17
One-at-a-Time Processing 18
How to Choose 19

5. Storage 21
Storage as the Fast Data Borders 21
The Message Backbone as Transition Point 22

6. Serving **23**
Sharing Stateful Streaming State 24
Data-Driven Microservices 24
State and Microservices 25

7. Substrate **27**
Deployment Environments for Fast Data Apps 28
Application Containerization 28
Resource Scheduling 29
Apache Mesos 29
Kubernetes 30
Cloud Deployments 30

8. Conclusions **33**

Introduction

We live in a digital world. Many of our daily interactions are, in personal and professional contexts, being proxied through digitized processes that create the opportunity to capture and analyze messages from these interactions. Let's take something as simple as our daily cup of coffee: whether it's adding a *like* on our favorite coffee shop's Facebook page, posting a picture of our latte macchiato on Instagram, pushing the Amazon Dash Button for a refill of our usual brand, or placing an online order for Kenyan coffee beans, we can see that our coffee experience generates plenty of events that produce direct and indirect results.

For example, pressing the Amazon Dash Button sends an event message to Amazon. As a direct result of that action, the message is processed by an order-taking system that produces a purchase order and forwards it to a warehouse, eventually resulting in a package being delivered to us. At the same time, a machine learning model consumes that same message to add *coffee* as an interest to our user profile. A week later, we visit Amazon and see a new suggestion based on our coffee purchase. Our initial single push of a button is now persisted in several systems and in several forms. We could consider our purchase order as a direct transformation of the initial message, while our machine-learned user profile change could be seen as a sophisticated aggregation.

To remain competitive in a market that demands real-time responses to these digital pulses, organizations are adopting Fast Data applications as a key asset in their technology portfolio. This application development is driven by the need to accelerate the extraction of value from the data entering the organization. The streaming

workloads that underpin Fast Data applications are often complementary to or work alongside existing batch-oriented processes. In some cases, they even completely replace legacy batch processes as the maturing streaming technology becomes able to deliver the data consistency warranties that organizations require.

Fast Data applications take many forms, from streaming ETL (extract, transform, and load) workloads, to crunching data for online dashboards, to estimating your purchase likelihood in a machine learning–driven product recommendation. Although the requirements for Fast Data applications vary wildly from one use case to the next, we can observe common architectural patterns that form the foundations of successful deployments.

This report identifies the key architectural characteristics of Fast Data application architectures, breaks these architectures into functional blocks, and explores some of the leading technologies that implement these functions. After reading this report, the reader will have a global understanding of Fast Data applications; their key architectural characteristics; and how to choose, combine, and run available technologies to build resilient, scalable, and responsive systems that deliver the Fast Data application that their industry requires.

CHAPTER 1

The Anatomy of Fast Data Applications

Nowadays, it is becoming the norm for enterprises to move toward creating data-driven business-value streams in order to compete effectively. This requires all related data, created internally or externally, to be available to the right people at the right time, so real value can be extracted in different forms at different stages—for example, reports, insights, and alerts. Capturing data is only the first step. Distributing data to the right places and in the right form within the organization is key for a successful data-driven strategy.

A Basic Application Model

From a high-level perspective, we can observe three main functional areas in Fast Data applications, illustrated in Figure 1-1:

Data sources
: How and where we acquire the data

Processing engines
: How to transform the incoming raw data in valuable assets

Data sinks
: How to connect the results from the stream analytics with other streams or applications

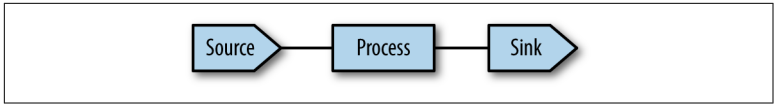

Figure 1-1. High-level streaming model

Streaming Data Sources

Streaming data is a potentially infinite sequence of data points, generated by one or many sources, that is continuously collected and delivered to a consumer over a transport (typically, a network).

In a data stream, we discern individual messages that contain records about an interaction. These records could be, for example, a set of measurements of our electricity meter, a description of the clicks on a web page, or still images from a security camera. As we can observe, some of these data sources are distributed, as in the case of electricity meters at each home, while others might be centralized in a particular place, like a web server in a data center.

In this report, we will make an abstraction of how the data gets to our processing backend and assume that our stream is available at the point of ingestion. This will enable us to focus on how to process the data and create value out of it.

Stream Properties

We can characterize a stream by the number of messages we receive over a period of time. Called the *throughput* of the data source, this is an important metric to take into consideration when defining our architecture, as we will see later.

Another important metric often related to streaming sources is *latency*. Latency can be measured only between two points in a given application flow. Going back to our electricity meter example, the time it takes for a reading produced by the electricity meter at our home to arrive at the server of the utility provider is the *network latency* between the edge and the server. When we talk about latency of a streaming source, we are often referring to how fast the data arrives from the actual producer to our collection point. We also talk about *processing latency*, which is the time it takes for a message to be handled by the system from the moment it enters the system, until the moment it produces a result.

From the perspective of a Fast Data platform, streaming data arrives over the network, typically terminated by a scalable adaptor that can persist the data within the internal infrastructure. This capture process needs to scale up to the same throughput characteristics of the streaming source or provide some means of feedback to the originating party to let them adapt their data production to the capacity of the receiver. In many distributed scenarios, adapting by the originating party is not always possible, as edge devices often consider the processing backend as always available.

Once the event messages are within the backend infrastructure, streaming flow control such as Reactive Streams can provide bidirectional signaling to keep a series of streaming applications working at their optimum load.

The amount of data we can receive is usually limited by how much data we can process and how fast that process needs to be to maintain a stable system. This takes us to the next architectural area of our interest: processing engines.

Processing Engines

The processing area of our Fast Data architecture is the place where business logic gets implemented. This is the component or set of components that implements the streaming transformation logic specific to our application requirements, relating to the business goals behind it.

When characterized by the methods used to handle messages, stream processing engines can be classified into two general groups:

One-at-a-time
: These streaming engines process each record individually, which is optimized for latency at the expense of either higher system resource consumption or lower throughput when compared to micro-batch.

Micro-batch
: Instead of processing each record as it arrives, micro-batching engines group messages together following certain criteria. When the criteria is fulfilled, the batch is closed and sent for

execution, and all the messages in the batch undergo the same series of transformations.

Processing engines offer an API and programming model whereby requirements can be translated to executable code. They also provide warranties with regards to the data integrity, such as no data loss or seamless failure recovery. Processing engines implement data processing semantics that relate how each message is processed by the engine:

At-most-once
: Messages are only ever sent to their destination once. They are either received successfully or they are not. At-most-once has the best performance because it forgoes processes such as acknowledgment of message receipt, write consistency guarantees, and retries—avoiding the additional overhead and latency at the expense of potential data loss. If the stream can tolerate some failure and requires very low latency to process at a high volume, this may be acceptable.

At-least-once
: Messages are sent to their destination. An acknowledgement is required so the sender knows the message was received. In the event of failure, the source can retry to send the message. In this situation, it's possible to have one or more duplicates at the sink. Sink systems may be tolerant of this by ensuring that they persist messages in an idempotent way. This is the most common compromise between at-most-once and exactly-once semantics.

Exactly-once [processing]
: Messages are sent once and only once. The sink processes the message only once. Messages arrive only in the order they're sent. While desirable, this type of transactional delivery requires additional overhead to achieve, usually at the expense of message throughput.

When we look at how streaming engines process data from a macro perspective, their three main intrinsic characteristics are scalability, sustained performance, and resilience:

Scalability
: If we have an increase in load, we can add more resources—in terms of computing power, memory, and storage—to the processing framework to handle the load.

Sustained performance
: In contrast to batch workloads that go from launch to finish in a given timeframe, streaming applications need to run 365/24/7. Without any notice, an unexpected external situation could trigger a massive increase in the size of data being processed. The engine needs to deal gracefully with peak loads and deliver consistent performance over time.

Resilience
: In any physical system, failure is a question of *when* and not *if*. In distributed systems, this probability that a machine fails is multiplied by all the machines that are part of a cluster. Streaming frameworks offer recovery mechanisms to resume processing data in a different host in case of failure.

Data Sinks

At this point in the architecture, we have captured the data, processed it in different forms, and now we want to create value with it. This exchange point is usually implemented by storage subsystems, such as a (distributed) file system, databases, or (distributed) caches.

For example, we might want to store our raw data as records in "cold storage," which is large and cheap, but slow to access. On the other hand, our dashboards are consulted by people all over the world, and the data needs to be not only readily accessible, but also replicated to data centers across the globe. As we can see, our choice of storage backend for our Fast Data applications is directly related to the read/write patterns of the specific use cases being implemented.

CHAPTER 2

Dissecting the SMACK Stack

In Chapter 1, we discussed the high-level characteristics of a Fast Data platform and applications running on top of it. In each of the three areas—data sources, processing, and data sinks—trade-offs are introduced by the various technologies available today. How do we make the right choices? To answer that question, let's explore a successful Fast Data application architecture.

The SMACK Stack

The *SMACK stack* is Spark, Mesos, Akka, Cassandra, and Kafka:

S: Spark
: A distributed processing engine capable of batch and streaming workloads

M: Mesos
: A cluster manager, also known as a "scheduler" that abstracts out resources from applications

A: Akka
: A highly concurrent and distributed toolkit for building message-driven applications

C: Cassandra
: A distributed, highly scalable, table-oriented NoSQL database

K: Kafka
: A streaming backend based on a distributed commit log

The SMACK stack is a distributed, highly scalable platform for building Fast Data applications based on business-friendly open source technologies. To learn more about the SMACK stack, Patrick McFadin's post is a good starting point.

The SMACK stack became popular because it offered an integration blueprint for implementing a Fast Data architecture, using open source components that shared similar scalability characteristics.

Their common denominator is *distributed*. Distributed partitions of a Kafka topic could be consumed by Spark tasks running in parallel in several executor nodes. In turn, Spark could write to Cassandra nodes, taking into account optimal micro batching based on key allocation. Akka actors could implement service logic by independently retrieving data from Cassandra or pushing new messages to Kafka by using an event-segregation model.

All these components run reliably on Apache Mesos, which takes care of scheduling jobs close to the data and ensures proper allocation of cluster resources to different applications.

As we have learned, there is no silver bullet. Although the SMACK stack is great at handling Internet of Things (IoT)/time series and similar workloads, other alternatives might better meet the challenges posed by current machine learning, model-serving, and personalization use cases, among others.

Functional Composition of the SMACK Stack

We want to extract that winning formula that makes the SMACK stack a success in its space. Taking a bird's-eye view of such architecture, we can identify the roles that these components play. By decomposing these roles into functional areas, we come up with the functional components of a Fast Data platform, shown in Figure 2-1.

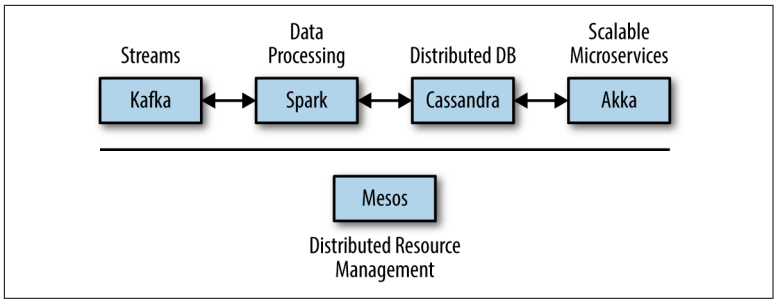

Figure 2-1. The SMACK stack functional components

For this report, we want to focus on the roles that each component is fulfilling in the architecture and map those roles to our initial requirements:

Message backbone
: How to ingest and distribute the data where we need it?

Compute engines
: How to transform raw data into valuable insights?

Storage
: How to persist data over time in a location where other applications can consume it?

Serving
: How to create data-powered applications?

Substrate
: Where and how do we run all this—and keep it running?

In the next chapters, we cover these functional areas in detail.

CHAPTER 3
The Message Backbone

The *message backbone* is a critical *subsystem* of a Fast Data platform that connects all its major components together. If you think about the message backbone as a nervous system, you can consider events as the triggers of electrical messages that travel back and forth across that system. The message backbone is the medium through which messages are sent and received from various sensors, data repositories, and data processors.

So what is an *event*?

> An event can be defined as "a significant change in state." For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold." A car dealer's system architecture may treat this state change as an event whose occurrence can be made known to other applications within the architecture. From a formal perspective, what is produced, published, propagated, detected, or consumed is a (typically asynchronous) message called the event notification, and not the event itself, which is the state change that triggered the message emission. Events do not travel; they just occur.
>
> —Event-driven architecture, Wikipedia

We can take away two important facts from this definition. The first is that event messages are generally asynchronous. The message is sent to signal observers that something has happened, but the source is not responsible for the way observers react to that information. This implies that the systems are decoupled from one another, which is an important property when building distributed systems.

Second, when the observer receives an event message in a stream, we are looking at the state of a system in the past. When we continuously stream messages, we can re-create the source over time or we can choose to transform that data in some way relative to our specific domain of interest.

Understanding Your Messaging Requirements

Understanding best practices for designing your infrastructure and applications depends on the constraints imposed by your functional and nonfunctional requirements. To simplify the problem, let's start by asking some questions:

Where does data come from, and where is it going?
: All data platforms must have data flowing into the system from a source (ingest) and flowing out to a sink (egress). How do you ingest data and make it available to the rest of your system?

How fast do you need to react to incoming data?
: You want results as soon as possible, but if you clarify the latency requirements you actually need, then you can adjust your design and technology choices accordingly.

What are your message delivery semantics?
: Can your system tolerate dropped or duplicate messages? Do you need each and every message exactly once? Be careful, as this can potentially have a big impact in the throughput of your system.

How is your data keyed?
: How you key messages has a large impact on your technology choices. You use keys in distributed systems to figure out how to distribute (partition), the data. We'll discuss how partitioning can affect our requirements and performance.

Let's explore some of the architectural decisions based on your answers.

Data Ingestion

Data ingestion represents the source of all the messages coming into your system. Some examples of ingestion sources include the following:

- A user-facing RESTful API that sits at the periphery of our system, responding to HTTP requests originating from our end users
- The Change Data Capture (CDC) log of a database that records mutation operations (Create/Update/Delete)
- A filesystem directory from which files are read

The source of messages entering your system is not usually within your control. Therefore, we should persist messages as soon as possible. A robust and simple model is to persist all messages onto an append-only event log (aka *event store* or *event journal*).

The event log model provides maximum versatility to the rest of the platform. Immutable messages are appended to the log. This allows us to scale the writing of messages for a few reasons. We no longer need to use blocking operations to make a write, and we can easily partition our writes across many physical machines to increase write throughput. The event log becomes the source of truth for all other data models (a golden database).

To create derivative data models, we replay the event log and create an appropriate data model for our use case. If we want to perform fast analytical queries across multiple entities, we may choose to build an OLAP cube. If we want to know the latest value of an entity, we could update an in-memory database. Usually, the event log is processed immediately and continuously, but that does not prevent us from also replaying the log less frequently or on demand with very little impact to our read and write throughput to the event log itself.

What we've just described is the Event Sourcing design pattern, illustrated in Figure 3-1, which is part of a larger aggregate of design patterns called Command and Query Responsibility Segregation (CQRS) and commonly used in event-driven architectures.

Event logs are also central to the Kappa architecture. Kappa is an evolution of the Lambda architecture, but instead of managing a batch and fast layer, we implement only a fast layer that uses an event log to persist messages.

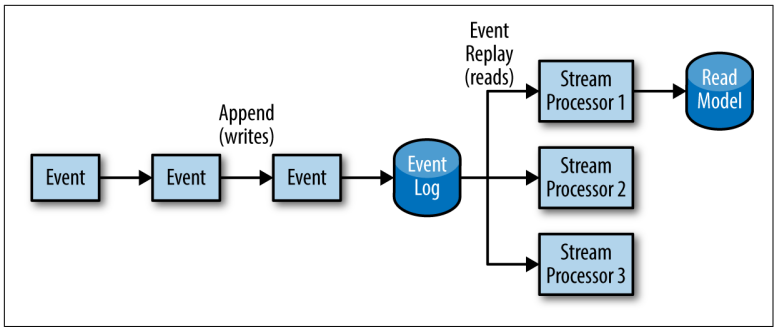

Figure 3-1. Event Sourcing: event messages append to an event log and are replayed to create different models

Apache Kafka is a Publish/Subscribe (or pub/sub) system based on the concept of a distributed log. Event messages are captured in the log in a way that ensures consumers can access them as soon as possible while also making them durable by persisting them to disk. The distributed log implementation enables Kafka to provide the guarantees of durability and resilience (by persisting to disk), fault tolerance (by replication), and the replay of messages.

Fast Data, Low Latency

How fast should Fast Data be? We will classify as Fast Data the platforms that can react to event messages in the millisecond-to-minutes range.

Apache Kafka is well suited for this range of latency. Kafka made a fundamental design choice to take advantage of low-level capabilities of the OS and hardware to be as low latency as possible. Messages produced onto a topic are immediately stored in the platform's Page Cache, which is a special area of physical system memory used to optimize disk access. Once a message is in Page Cache, it is queued to be written to disk and made available to consumers at the same time. This allows messages passing through a Kafka broker to be made available nearly instantaneously to downstream consumers because they're not copied again to another place in memory or buffer. This is known as *zero-copy transfer* within the Kafka broker.

Zero-copy does not preclude the possibility of streaming messages from an earlier offset that's no longer in memory, but obviously this operation will incur a slight delay to initially seek the data on disk

and make it available to a consumer by bringing it back into Page Cache. In general, the most significant source of latency when subscribing to a Kafka topic is usually the network connection between the client and broker.

Kafka is fast, but other factors can contribute to latency. An important aspect is the choice of delivery guarantees we require for our application. We discuss this in more detail in the next section.

Message Delivery Semantics

As we saw in Chapter 1, there are three types of message delivery semantics: at-most-once, at-least-once, and exactly-once. In the context of the message backbone, these semantics describe how messages are delivered to a destination when accounting for common failure use cases such as network partitions/failures, producer (source) failure, and consumer (sink, application processor) failure.

Some argue that exactly-once semantics are impossible. The crux of the argument is that such delivery semantics are impossible to guarantee at the protocol level, but we can fake it at higher levels. Kafka performs additional operations at the application processing layer that can fake exactly-once delivery guarantees. So instead of calling it exactly-once message delivery, let's expand the definition to exactly-once processing at the Application layer.

A plausible alternative to exactly-once processing in its most basic form is at-least-once message delivery with effective idempotency guarantees on the sink. The following operations are required to make this work:

- Retry until acknowledgement from sink.
- Idempotent data sources on the receiving side. Persist received messages to an idempotent data store that will ensure no duplicates, or implement de-duplication logic at the application layer.
- Enforce that source messages are not processed more than once.

Distributing Messages

A topic represents a type of data. Partitioning messages is the key to supporting high-volume data streams. A *partition* is a subset of messages in a topic. Partitions are distributed across available Kafka

brokers, as illustrated in Figure 3-2. How you decide which partition a message is stored in depends on your requirements.

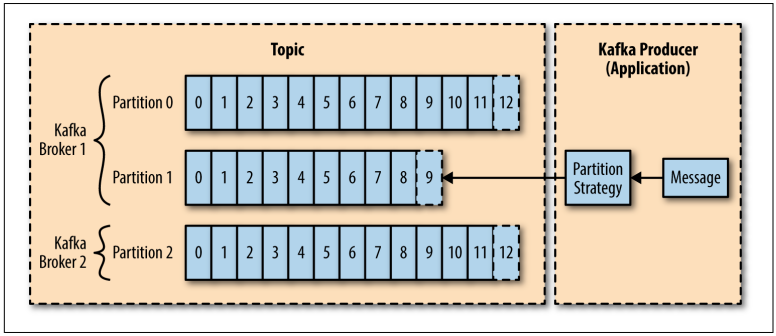

Figure 3-2. Kafka topic partitioning strategy

If your intention is to simply capture discrete messages and *order does not matter*, then it may be acceptable to evenly distribute messages across partitions (round-robin), similar to the way an HTTP load balancer may work. This provides the best performance as messages are evenly distributed. A caveat to this approach is that we sacrifice message order: one message may arrive before another, but because they're in two different partitions being read at different rates, an older message might be read first.

Usually, we decide on a partition strategy to control the way messages are distributed across partitions, which allows us to maintain order with respect to a particular key found in the message. Partitioning by key allows for horizontal scalability while maintaining guarantees about order.

The hard part is choosing what key to use. It may not be enough to simply pick a unique identifier, because if we receive an uneven distribution of messages based on a certain key, then some partitions are busier than others, potentially creating a bottleneck. This problem is known as a *hot partition* and generally involves tweaking your partitioning strategy after you start learning the trends of your messages.

CHAPTER 4
Compute Engines

At the center of a Fast Data architecture, we find *compute engines*. These engines are in charge of transforming the data flowing into the system into valuable insights through the application of business logic encoded in their specific model. As we learned in Chapter 1, there are two main stream processing paradigms: micro-batch and one-at-a-time message processing.

Micro-Batch Processing

Micro-batching refers to the method of accumulating input data until a certain threshold, typically of time, in order to process all those messages together. Compare it to a bus waiting at a terminal until departure time. This bus is able to deliver many passengers to their destination who are sharing the same transport and fuel.

Micro-batching enjoys *mechanical sympathy* with the network and storage systems, where it is optimal to send packets of certain sizes that can be processed all at once.

In the micro-batch department, the leading framework is Apache Spark. Apache Spark is a general-purpose distributed computing framework with libraries for machine learning, streaming, and graph analytics.

Spark provides high-level *structured* abstractions that let us operate on the data viewed as records that follow a certain schema. This concept ties together a high-level API that offers bindings in Scala, Java, Python, and R with a low-level execution engine that translates

the high-level structure-oriented constructs into query and execution plans that can be optimally pushed toward the data sources.

With the recent introduction of Structured Streaming, Spark aims at unifying the data analytics model for batching and streaming. The streaming case is implemented as a recurrent application of the different queries we define on the streaming data, and enriched with event-time support, windows, different output modes, and triggers to differentiate the ingest interval from the time that the output is produced.

One-at-a-Time Processing

One-at-a-time message processing ensures that each message is processed as soon as it arrives in the engine; hence, it delivers results with minimal delay. At the same time, shipping small messages individually increases the overall overhead of the system and therefore reduces the number of messages we can process per unit of time. Following the transportation analogy, one-at-a-time processing is like a taxi: an individual transport that can take a single passenger to its destination as fast as possible. (We are imagining here a crazy NYC driver for the lowest latency possible!)

The leading engine in this category is Apache Flink. Flink is a one-at-a-time streaming framework, also offering snapshots to isolate results from machine failure. This comprehensive framework provides a lower-level API than Structured Streaming, but it is a competitive alternative to Spark Streaming when low-latency is the key focus of interest. Flink presents APIs in Scala and Java.

In this space, we find Kafka Streams and Akka Streams. These are not frameworks, but libraries that can be used to build data-oriented applications with a focus on data analytics. Both offer low-latency, one-at-a-time message processing. Their APIs include projections, grouping, windows, aggregations, and some forms of joins. While Kafka Streams comes as a standalone library that can be integrated into applications, Akka Streams is part of the larger Reactive Platform with a focus on microservices.

In this category, we also find Apache Beam, which provides a high-level model for defining parallel processing pipelines. This definition is then executed on a *runner*, the Apache Beam term for an execution engine. Apache Beam can use Apache Apex, Apache

Flink, Apache Gearpump, and the proprietary Google Cloud Dataflow.

How to Choose

The choice of a processing engine is largely driven by the throughput and latency requirements of the use case at hand. If we need the lowest response time possible—for example, an industrial sensor alert, alarms, or an anomaly detection system—then we should look into the one-at-a-time processing engines.

If, on the other hand, we are implementing a massive ingest and the data needs to be processed in different data lines to produce, for example, a registration of every record and aggregated reports, as well as train a machine learning model, a micro-batch system will be best suited to handle the workload.

In practice, we observe that this choice is also influenced by the existing practices in the enterprise. Preferences for specific programming languages and DevOps processes will certainly be influential in the selection process. While software development teams might prefer compiled languages in a stricter CI/CD pipeline, data science teams are often driven by availability of libraries and individual language preferences (R versus Python) that create challenges on the operational side.

Luckily, general-purpose distributed processing engines such as Apache Spark offer bindings in different languages, such as Scala and Java for the discerning developer, and Python and R for the data science practitioner.

CHAPTER 5
Storage

In many cases, when we refer to *Big Data*, we usually relate it to a large storage infrastructure. In the past decade, when Hadoop-based architectures became popular, the challenge that they were solving was twofold: how to reliably store large amounts of data and how to process it. The Hadoop File System (HDFS), with its concept of replicated blocks, provided reliability in case of hardware failure, while MapReduce brought parallel computations to where the data was stored to remove the overhead of moving data over the network. That model is based on the premise that the data is already "at rest" in the storage system.

Storage as the Fast Data Borders

In the particular case of Fast Data architectures, storage usually demarks the transition boundaries between the Fast Data core and the traditional applications that might consume the produced data. The choice of storage technology is driven by the particular requirements of this transition between moving and resting data.

If we need to store the complete data stream as it comes in, and we need access to each individual record or sequential slices of them, we need a highly scalable backend with low-latency writes and key-based query capabilities. As we learned in Chapter 2, Apache Cassandra is a great choice in such a scenario, as it offers linear scalability and a limited but powerful query language (Cassandra Query Language, or CQL).

Data could also then be loaded into a traditional data warehouse or could be used to build a data lake that can support different capabilities including machine learning, reporting, or ad hoc analysis.

On the other side of the spectrum, we have predigested aggregates that are requested by a frontend visualization system. Here, we probably want the full SQL query and indexing support to quickly locate those records for display. A more classical PostgreSQL, MySQL, or the commercial Relational Data Base Management System (RDBMS) counterparts would be a reasonable choice.

Between these two cases is a whole range of options, ranging from specialized databases (such as InfluxDB for time series or Redis for fast in-memory lookups), to raw storage (such as on-premises HDFS) or the cloud storage offerings (Amazon S3, Azure Storage, Google Cloud Storage, and more).

The Message Backbone as Transition Point

In some cases, it is even possible to use the message backbone as the data hand-over point. We can exploit the capabilities of a persistent event log such as Apache Kafka, discussed in Chapter 3, to transition data between the Fast Data applications and clients with different runtime characteristics. A blog by Jay Kreps summarizes the particular set of use cases for which this is a reasonable option.

When dealing with storage choices, there is no one-size-fits-all. Every Fast Data application will probably require a specific storage solution for each integration path with other applications in the enterprise ecosystem.

CHAPTER 6
Serving

Fast Data applications are built to deliver continuous results and are consumed by other apps and microservices. Some examples include real-time dashboards for monitoring business Key Performance Indicators (KPIs), or an application to enrich the analytical capabilities of Business Intelligence (BI) software, or an aggregation of messages to be queried by a RESTful API. Applications may also apply machine learning (ML) techniques to the data, such as scoring an ML model, or even train a model on the fly.

Let's explore some patterns we can use in a serving layer. We can use a Big Table–based technology, such as Cassandra or HBase (or, more traditionally, RDBMS), that is continuously updated. Users then consume the data with client applications that read from them.

Importing batch data into highly indexed and aggregated data stores used with analytical data-mining BI tools is a common practice. A newer trend is to use Streaming SQL to apply analytical transformations on live data. Streaming SQL is supported by all major stream processors including Apache Spark, Apache Flink, and Kafka Streams.

Finally, it is possible to serve data directly from the message backbone. For example, we can consume messages from a Kafka topic into a dashboard to build a dynamic, low-latency web application.

Sharing Stateful Streaming State

When running stateful streaming applications, another possibility is to share a view of that state directly. This is a relatively new ability in stream processors. Some options available today are Flink Queryable State and interactive queries for Kafka Streams, including its akka-http implementation by Lightbend.

With respect to machine learning, it's possible to integrate state with machine learning models to facilitate scoring. For example, see the Flink Improvement Proposal (FLIP) for Model Serving.

Data-Driven Microservices

Just as our Fast Data applications are data-driven, so are microservices. In fact, implementing microservices in this way is not a new concept. If we drop the fashionable label of "microservices" and think of them as application services, then we've seen this pattern before with service-oriented architecture (SOA) and enterprise service bus (ESB).

We can link microservices in a similar fashion to using an ESB, but by using Apache Kafka as the message backbone instead. As an example, Figure 6-1 illustrates a simplified architecture for an e-commerce website that relies on Kafka as the messaging infrastructure that supports its message exchange model. By using Kafka, we can to scale our services to support very high volume as well as easily integrate with stream processors.

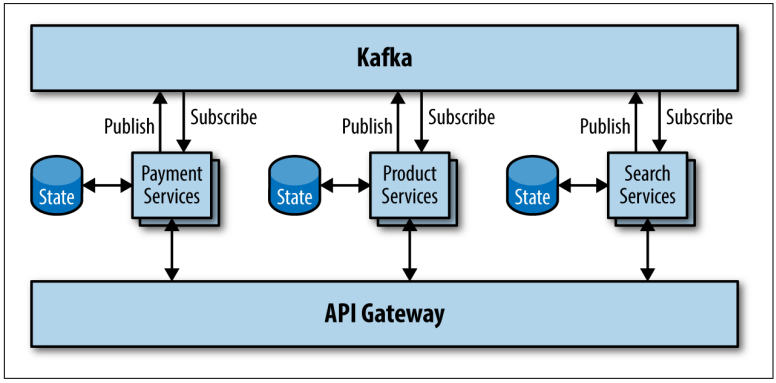

Figure 6-1. A simplified e-commerce example of a microservices architecture using Kafka as a message bus

In microservices, we promote nonblocking operations that reduce latency and overhead by asynchronously publishing and subscribing to event messages. A service subscribes to messages to get state changes to relevant domain entities, and publishes messages to inform other services of its own state changes. A service becomes more resilient by encapsulating its own state and becoming the gateway to accessing it. A service can stay online without failing as a consequence of dependent services going down. In such a case, the service will continue to function but may not be up-to-date.

Microservices share many of the same properties of a Fast Data application, to the point that it's becoming more difficult to distinguish them. Both stream unbounded sets of data in the form of subscribing to messages or listening for API calls. Both are always online. Both output something in the form of API responses or new messages to be subscribed to by yet other microservices or Fast Data applications.

The main distinction is that a microservice allows for general application development in which we're free to implement any logic we want, whereas a Fast Data application is implemented using a stream processor that may constrain us in various ways. However, Fast Data apps are becoming more generic as well. Therefore, we conclude that microservices and Fast Data applications are *converging* and often have the same domain requirements, use the same design patterns, and have the same operational experiences.

State and Microservices

Stream processors have various means to maintain state, but historically it's been challenging to provide a rich stateful experience. We've already mentioned that new libraries are available to share state in stream processors, but the technology is still in its early days, and developers are often forced to call out to more general-purpose storage.

Akka is a great way to model a complex domain and maintain state. Akka is a toolkit, not a framework, so you can bring components in when you need them to build highly customized general-purpose applications that can stream from Kafka (reactive-kafka), expose HTTP APIs (akka-http), persist to various databases (Akka Persistence, JDBC, etc.), and much more.

Akka Cluster provides the tools to distribute your state across more than one machine. It includes advanced cluster-forming algorithms and conflict-free replicated data types (CRDTs, similar to those used by various distributed databases), and can even straddle multiple data centers. Exploring Akka is a huge topic, but it's essential reading when you have a complex stateful requirement for your Fast Data platform.

CHAPTER 7
Substrate

Let's assume we have figured out the major components of our data architecture, and the pieces start to fit in the puzzle that will serve the business use case. The next question is, how do we deal with the infrastructure layer to support those components? Our base requirement is that we need it to run our systems/components in an efficient and cost-effective manner. We also require that this layer provides resource management, monitoring, multitenancy, easy scaling, and other crucial operational capabilities in order to implement our architecture on top of it.

As usual in computer science, the solution is to use an additional abstraction layer. This infrastructure abstraction, let's call it the *substrate*, allows us to run a wide variety of software components while providing several core operational capabilities. At its core, this layer is essentially an abstraction on top of hardware resources and operating system functions in a distributed setting. And as in an operating system, we want this layer to provide a set of basic services to the application running on top of it:

- Allocate enough resources as needed that are fairly distributed among applications
- Provide application-level isolation to securely run applications from different business owners
- Ensure application resilience in case of failure of the underlying hardware

- Expose usage metrics to enable system operators to decide on capacity planning
- Provide management and monitoring interfaces to integrate its operational aspects into the rest of the enterprise system

Cluster managers such as Yarn and Mesos emerged to solve this problem, while container orchestrators such as Kubernetes provide additional facilities to easily deploy and operate applications.

Deployment Environments for Fast Data Apps

Fast Data applications create high expectations because the idea of delivering insights or personalized user-experiences almost in real time is appealing. Systems running these applications have to be available all the time, need to be scalable, and must maintain a stable performance while processing large volumes of data at scale. The latter requires us to rethink our infrastructure and how we deploy our applications. There is a clear need for a solution that embraces distributed computing at its core.

Two complementary technology concepts are driving this area: containerization of applications and efficient scheduling of resources. Let's discuss these two areas in more detail.

Application Containerization

Containerization is a technology that allows running applications in an isolated manner within a host. We can compare it with virtual machines (VMs), which have been the leading hardware virtualization technology of the past decade. When compared to VMs, containers are said to be *lightweight*. That is, instead of building upon a full operating system, containers use Linux isolation technologies:

Linux namespaces
: These ensure that each process sees only its own limited view of the system. This provides isolation from other processes.

Linux control groups (aka c-groups)
: These are used to limit the amount of machine resources (CPU, memory, disk) that a "containerized" process can consume. This enables fine-grained resource management, such as assigning 0.5 CPUs to a process, and improves overall resource utilization.

Docker, which popularized the container technology, combined these two *ingredients* with a portable image format to enable ease of distribution and reproducible deployments across different hosts. This facilitates a DevOps practice, in which application developers can package their software in an image that can run on their laptop for development and testing, or on production to handle a real load.

Lowering the development-to-deployment barrier increases enterprise agility and its ability to respond to the changing environment. This means Fast Data applications that are *fast* not only because of their streaming data affinity, but also from the development and deployment perspective, leading to an increased business agility.

Resource Scheduling

Assuming that you have containerized your stateful/stateless apps, allocating resources (CPU, memory, disk, network) and running them is the next step. This is where the cluster manager comes into play.

The approach is to split the concerns of resource allocation, task scheduling, and execution on nodes. This approach is called *two-level scheduling* (Mesos, Yarn to some extent) and allows for applications to develop their own scheduling policy by moving scheduling logic to the application code while resource allocation is done by the scheduler.

Apache Mesos

Apache Mesos introduces a modular architecture in which applications can bring their own scheduler while Mesos takes care of the resource allocation through a resource offer model. Applications are offered resources such as CPU or memory, or even specialized capabilities such as GPUs. By accepting or rejecting these resources, applications can fulfill their specific needs, while Mesos remains agnostic of the workloads. This allows Mesos to colocate diverse workloads, such as databases, computing engines, and microservices on the same shared infrastructure.

Apache Mesos and its commercially supported distribution, Mesosphere DC/OS, offers a single environment for operating both microservices and open source data services to support all the components of Fast Data applications. It supports several data services

such as Kafka and HDFS, and data-oriented frameworks such as Apache Spark and Apache Flink out of the box, through its package manager. Also, more traditional services can be found listed, including MySQL, PostgreSQL, and others. Services and applications run on the same environment and benefit from features like a unified approach for scalability, security, deployment, and monitoring.

Kubernetes

On the container scheduling side, we find Kubernetes, which is the open source project with the fastest growing popularity. Kubernetes is focused on container orchestration, and it has a massive community around its open source development, hosted by the Cloud Native Computing Foundation. Kubernetes has its foundations in two previous systems for scheduling applications at Google: Borg and its successor, Omega. As such, it builds upon more than a decade of experience in running one of the largest computing loads in the world. Kubernetes supports several services and key features for deploying and operating applications. Some features it has are unique—for example, its federation capability that allows you to handle multiple clusters across data centers. On the other hand, support of different data-centric frameworks on Kubernetes is currently in the early stages and under active development. In particular, Apache Spark recently released official support for Kubernetes in its 2.3.0 version.

Kubernetes has its own philosophy, which is strongly influencing the container orchestration movement.

Last but not least, both Apache Mesos and Kubernetes have been proven in production, supporting large clusters of thousands of nodes. Kubernetes can be run as one of the container orchestration options on top of Mesosphere DC/OS.

Cloud Deployments

So far, we have talked about technologies that are not specific to any cloud infrastructure or on-premises hardware. While it is possible to select technologies to implement a platform for Fast Data applications by using cloud vendor–specific technologies, staying cloud neutral has a real benefit. Technologies such as DC/OS and Kubernetes provide the tools you need out of the box to build your Fast

Data applications while also allowing you to avoid vendor lock-in. Both technologies can run natively on most existing cloud environments. In summary, what you gain is the flexibility to dynamically run workloads where resources are available.

CHAPTER 8
Conclusions

With Fast Data applications, we understand the domain of data-intensive applications that aim to continuously process and extract insights from data as it flows into the system. Fast Data architectures define the set of integrated components that provide the building blocks to create, deploy, and operate scalable, performant, and resilient applications around the clock.

Using the SMACK stack as a blueprint of a successful Fast Data architecture, we identified the key functional areas and how they integrate into a consistent platform:

Message backbone
: This component is responsible for ingesting and distributing data among the components within the Fast Data boundaries. Apache Kafka is the leading project in this area. It delivers a publish/subscribe model backed by a distributed log abstraction that provides the guarantees of durability, resilience, fault tolerance, and the ability to replay messages by different consumers.

Compute engines
: This is the place where business logic gets applied to the data. Choosing the right engine is driven by the application requirements, with throughput and latency as key discriminators. Other influencing factors include the supported languages versus the target users and its application domain. Some engines, including Apache Spark and Apache Flink, can also be used for general data processing, while Kafka Streams and Akka Streams

might be better choices when it comes to integrating microservices into the Fast Data architecture.

Storage
: Storage subsystems form the ideal transition point between the Fast Data domain and client applications. They act as buffers between the data in motion delivered by the Fast Data applications and external clients with different runtime characteristics. The choice of a storage system is usually bound to each application and driven by the write and read patterns it presents.

Serving
: The serving infrastructure is usually bound to the storage system and typically offers data as a service. It provides a layer of convergence of Fast Data and microservices, where microservices make sense of data and present it to their specific domain. This could range from HTTP/REST endpoints offering a view on the data to machine learning model serving, in which the Fast Data component updates the model information with fresh data, while the serving layer presents a stable interface to external clients.

Substrate
: This is the infrastructure abstraction that provides resources to services, frameworks, and applications in a secure, monitorable, and resilient way. Containerization technology creates self-contained reproducible deployment units. These containers can be orchestrated by cluster managers to ensure that the applications they contain get their required resources, are restarted in case of failure, or relocated when the underlying hardware fails. Apache Mesos with the DC/OS distribution and Kubernetes are the leading container orchestrators.

We are spoiled with choices. Some of them are more obvious than others. The challenge for software architects is to match their application and business requirements with the range of options available to make the right decisions at every layer of the architecture.

A successful implementation of the Fast Data architecture will deliver the business the ability to develop, deploy, and operate applications that provide real-time insights and immediate actions, increasing its competitive advantage and agility to react to specific market challenges.

About the Authors

Stavros Kontopoulos is a distributed systems engineer, interested in data processing, programming, and all aspects of computer science. He is currently working as a senior software engineer on the Fast Data Platform team at Lightbend. When not busy with technology, he enjoys traveling and working out.

Sean Glover is a software architect, engineer, teacher, and mentor. He's an engineer on the Fast Data Platform team at Lightbend. Sean enjoys building streaming data platforms and reactive distributed systems, and contributing to open source projects.

Gerard Maas is a seasoned software engineer and creative soul with a particular interest in streaming architectures. He currently contributes to the engineering of the Fast Data Platform at Lightbend, where he focuses on the integration of stream processing technologies. He is the coauthor of *Stream Processing with Apache Spark* (O'Reilly).