Building a Segment-Inspired Data Pipeline Platform Using Apache Kafka, Spark and Node.js on AWS

Andreja Nesic
School of Computing
Union University
11000 Belgrade, Republic of Serbia
anesic3119rn@raf.rs

Jan 2023, Belgrade, Serbia

1 Abstract

In today's interconnected world, building an isolated software ecosystem is becoming increasingly rare. As businesses grow, so does their need to integrate with partners' systems or to create separate units inside the organization which can function standalone. This results in the need for an underlying communication infrastructure for all software involved in the process. If done from scratch, creating that infrastructure requires a significant investment, both financially and time-wise. According to Wakefield Research, 44% of data engineers' time is spent manually building and managing data pipelines, resulting in an average cost of $519,522 per year [1]. Furthermore, such an infrastructure requires constant maintenance and adapting as the systems it connects evolve, and requirements change.

In this paper, we present an idea based on the Segment CDP platform [2] of creating a standardized data pipeline for event-driven software integration in the context of small businesses, with a simple user interface which abstracts the details of the implementation of connections from the user. This idea enables organizations to integrate with partners or create internal data pipelines with a limited amount of technical know-how. We will also examine why Apache Kafka and Spark are among the best tools to apply to this problem.

2 Table of Contents

3 Introduction

According to a report by Forbes in 2019, 95% of businesses need to manage some sort of unstructured data, with over 40% saying this was done on a frequent basis [3]. As more and more organizations discover the benefits of big data, analytics and AI, the need for data pipelines will only continue to grow. What's more, as small and medium businesses look to harness the benefits of data analytics, the market will need simple and cost-effective solutions to keep up with the demand—according to Wakefield Research, 44% of data engineers' time is spent manually building and managing data pipelines, resulting in an average cost of $519,522 per year [1]. Such an infrastructure also incurs costs from constant maintenance and adapting as the systems it connects evolve, and requirements change. Building a data pipeline with this sort of time and budget is simply not possible for many SMB organizations. Thus, it is reasonable to conclude that a simplified, yet versatile data pipeline could present a unique market opportunity, while helping organizations that are beginning to adopt big data practices.

Today, data is being generated at a rapid pace, and new technologies are putting pressure on big data systems to continuously evolve. Let's examine some of the challenges data pipelines are facing today:

- **Exponential growth.** More and more information is being generated from a vast array of sources, such as financial markets, healthcare systems, traffic and transportation industry, handheld devices, social networks, and more. The sheer volume of data means data pipelines need lots of resources to meet the demand, or they could risk losing information and producing skewed insights.
- **Heterogeneous sources.** Producers of data differ vastly in their protocol and channel of choice for delivering data. Connecting with each system poses a technical challenge and can result in complex, overly-coupled architecture if not executed well.
- **Data context and transformation.** Most events, such as traffic data, market movements, weather changes, user decisions, and more, need to be analyzed in their context. When inspected in isolation, they provide little insights. But analyzed over a certain time window, and with the right transformations, these data streams provide powerful knowledge. Defining these transformations and setting rules for data analysis is of crucial importance to data pipelines, especially if their users lack technical know-how.
- **Real-time relevance.** As markets get more competitive, making the right decision at the right time is becoming crucial. This means businesses need new information as soon as possible, and legacy data systems often fail to produce on time. Modern data pipelines are required to deliver not just information, but also actionable analysis in (near) real-time, or else they may fail to provide relevant insights.
- **Security and maintenance.** Today's malware is ever-evolving, and new liabilities are being uncovered every day, even in the most trusted of libraries. Complex systems such as pipelines are prime targets for malicious actors due to the nature of their data. Thus, a data pipeline development team must take great care in building a robust, secure system, and ensure its security is updated to meet threat protection standards and decisions from the relevant lawmakers.

To conclude, big data systems are facing a growing set of challenges, posed not only by stakeholders, but also by external systems and consumers. Building an infrastructure that meets all these demands is difficult for organizations with little to no technical know-how; thus, a simple, general use-case data pipeline that solves the aforementioned issues, without requiring technical expertise from the end-users, could present a good market opportunity. We will thus define the exact problems our product will face from a technical perspective, and examine the available technology to solve them.

4 Problem Formulation

Our platform's key challenge is to enable the creation of reliable, scalable and maintainable data pipelines between external, user-defined systems, without requiring technical sophistication from the user. Several sub-issues can be recognized as part of this problem. We shall denote systems which send data to our platform for further processing and routing as data sources, and systems which receive the data at the other end of the pipeline as data consumers.

4.1. Data Sources

The platform must work with a wide array of connected data (event) sources. These sources of data will be sending periodic or constant streams of data towards the platform. Several key issues may be observed:

Uptime. As data sources represent 3rd party systems, we may not rely on them to resend unprocessed data due to the platform's unavailability. Thus, the platform must provide near-100% uptime at critical points in the system (such as producers) in order to avoid data loss. Another implication is the need for built-in fault tolerance, should a hardware or software component fail.

Geographic availability. Data sources are likely to be scattered geographically. In order to provide low-latency service to customers, our platform has to be available in multiple regions across the globe. This will minimize data travel time between sources and consumers, helping deliver data in (close to) real-time.

Protocol. The platform shall connect with data sources via their publicly-available APIs. These APIs may be available for access through different protocols (HTTP(S), WebSocket, RPC, etc.) Our platform will require a custom interface for each supported data source, which in turn means supporting the source's preferred protocol, whichever protocol that may be. (We shall mostly consider state-of-the-art protocols.)

Schema. Most data sources will vary widely in types of transmitted data, and its schema. The platform's source-facing interfaces must be able to properly read the received data, and, if required, decode and process it (optionally transforming it into another schema format) before forwarding the data to consumers.

Security and authentication. As the platform will be collecting data on behalf of its users, the connected platforms are likely to require authentication in order to establish a connection. This adds extra complexity to the system as it has to store authentication credentials in a secure manner, and cater for any extra authentication steps presented by data sources when establishing a connection.

4.2. Data Consumers

The platform will enable users to route incoming data into a multitude of receiving systems (consumers). The challenges of consumers are very similar to those related to data sources. However, data consumers also present a few unique issues:

Reception guarantee. One of the core promises of our platform is reliability—data will be transmitted from sources to consumers regardless of issues encountered in the connection. However, the consumer may not be able to process the data we are sending. For instance, the consumers' server may impose a cap on how much data we may transmit. Issues may also be encountered due to the network, or due to the consumers' internal faults. All this means our platform needs to keep track of responses received from the consumers (or lack thereof) and retry sending the data accordingly. Consequently, there may be a need to store message data temporarily until all consumers have confirmed its successful reception, or a limit in terms of data memory or storage time is reached.

At-least-once and exactly-once semantics. Different Consumers will impose different requirements in terms of data delivery. For example, it is critical that a payment processor does *not* receive the same charge twice, while an email marketing service will be affected very little if a customer click is recorded multiple times in error. Therefore, the semantics of at-least-once and exactly-once must be addressed by the platform. This means the platform will need built-in support for both systems, with configurable architecture and settings per sink.

Timeliness. Some systems send/read data in bulk at certain times, while others do so through a continuous flow. Our data pipeline platform must support all cases: systems for receiving data must maintain high availability at all times, no matter whether data arrives in bulk or through a stream; while consumers must be configured to service endpoint applications in bulk or via a stream, whichever is required.

4.3. Data Flow

Security and privacy. Our platform may be required to handle and distribute sensitive data, such as medical or financial information, personally-identifiable data, etc. This issue represents a major challenge from a security and legal perspective; users' data, and any data received on account of our users, must be stored soundly, using state-of-the-art encryption. The platform shall have to abide by the privacy and data protection laws of any country its servers are located in; it may also be required to abide by the respective data/privacy laws of the countries it is receiving/sending the data to, in addition to complying with practices requested by the provider of server hardware. Furthermore, systems sending/receiving data to/from our platform are likely to require authentication from the platform acting on the client's behalf, which is why the product will require a robust and safe credential management system.

Fault-tolerance. As a distributed system, our platform is expected to face hardware (and software) failures, network failures, and more, all of which have to be dealt with through the application architecture. It is imperative that fault tolerance, high availability and data redundancy are built into the system from the very start of the design process. This means the application has to be distributed across a number of nodes which will provide computational availability, and availability of redundant data; events must be replicated in the system to avoid data loss; the possibility of reprocessing and human error [4] must also be taken into account, as well as the prospect of errors caused by the development and maintenance teams.

4.4. Maintenance

Schema evolution. As systems connected to our platform evolve, so will their APIs. This represents a problem for our platform, as any change in the API schema implies the need to adapt the platform's interface for connecting to that system. This task can hardly be automated and will likely incur significant long-term costs for the platform. For this reason, the platform must segregate all of its components through separation of concerns/abstraction [5], in order to facilitate the development of future updates.

Access credentials. Due to the sensitive nature of data hosted on the platform, extreme caution must be exercised when developing access points for the development team/customer service team for gaining access to the platform's backend, so as to avoid potential privacy or security breaches. Credentials and permissions of the maintenance team must be programmatically restricted and minimized to the fewest permissions needed to operate the platform, resolve customer issues, and so forth.

Extreme processing [4]. In order to preserve forwards-compatibility, including the integration of new applications as sinks, our data pipeline must make minimal transformations (if any) to the original event data. In this manner, it is possible to manipulate the "raw" messages and avoid losing critical data in case it is needed downstream.

4.5. Other Issues

Other issues are expected to be encountered as more research is conducted and the platform development begins. Nevertheless, the key principles in designing the platform remain that of reliability, high availability (scalability), fault-tolerance, low latency, maintainability, inherent security and minimal technical requirements for the end-user.

5 Technology and Literature Review

Our problem belongs to the domains of big data, batch processing, ETL/ELT systems, cloud applications, distributed computing and more. These technologies are growing in importance, with a rapidly-expanding body of knowledge, best practices, and tools on offer, as more organizations discover their benefits. With so many recommended practices, frameworks, and programming paradigms, it is important to make the right choice for the requirements of our platform. We shall thus examine some of the key ideas, platforms and methodologies behind today's modern distributed big data applications and determine which may be suitable as components of our data pipelining solution. Special attention will be given to Apache Kafka, our data pipeline/streaming technology of choice, being the building block of our product.

5.1 Paradigms

Paradigms, or patterns, are ubiquitous in applied computer science. They are especially important in cloud and big data technologies, as building software in these domains without proven practices may introduce unwanted risks. While there are a host of paradigms present in today's cloud computing, the following are of prime significance for our solution:

5.1.1 Event-Driven Architecture

Event-Driven Architecture is a software architecture pattern/concept in which systems generate, transmit, and react to **Events.** An Event is considered to be any change in application state, such as the click of a button in a graphical user interface. Systems which generate Events are called **Producers.** The emitted event data is referred to as an **Event Notification,** and is used interchangeably with the term "Event". Events may contain data, such as the value of a transaction, or they may simply indicate that a change occurred, such as a user interaction. Our data pipeline platform will be based on the EDA paradigm as we anticipate receiving Events from external systems, with the goal of forwarding them to the selected receiving systems, in a user-configured manner. In EDA, Events pass through **Channels,** which represent "routes" the data flows through. Channels connect Producers to endpoints, called **Sinks** (or Event processing engines). Sinks represent consumers of the event notification—event data is processed by Sinks and may be forwarded for use by other systems [6].

Decoupling and distributed design. A key benefit of EDA is its decoupled nature [7]. Event Producers are oblivious to the existence of Consumers and vice versa. The two systems share no state and need not be synchronized. The only coupling existing between Producers and Consumers is the shared event notification schema [4], enabling the communication between these two otherwise unrelated systems. However, although there exists no coupling in theory, many issues arise when connecting Producers and Consumers in practice. These issues mostly lie in the logical connection of Producers and Consumers, and conflicts which arise when their behaviors diverge from the requirements of their counterpart. For example, a Consumer expecting a steady stream of data may not be able to deal with a batch of Events dispatched periodically by a Producer; errors may also be encountered if the Producer keeps sending Events

to the Consumer when the latter is unavailable; repeat messages sent by the Producer may cause errors in data consistency for the Consumer, and so forth. Some of these issues apply to any general data pipeline case and have been solved by popular frameworks, while others, such as conflict resolution, require a custom procedure.

5.1.2 Microservice Architecture and EDA

Modern software systems are largely built on the **Microservice architecture.** Here, the system consists of multiple decoupled components (**Services**) which communicate through APIs, and tend to have little overlap in computing results—there exists a separation of concerns in Services' functionality. For example, an e-commerce software may have a customer-facing web server component intended to receive orders; a product fulfillment service which dispatches ordered products; an email automation service that collects newly-acquired customer emails and sends important updates, and so forth. Each Service is designed, deployed and scaled independently, resulting in strong overall resilience of the system, and a decoupled architecture. Because of its decoupling, Event-Driven data pipelines are commonly built into the Microservice architectures. With EDA, each Service can be subscribed to a certain Event Channel to react to transmitted messages; each Service independently "reacts" to an Event distributed to the Channel. However, Microservices may themselves act as Producers of Events towards other Microservices in the solution, as is the case in a **Saga architecture** [8]. In the previous example, the customer-facing website may generate events towards the product fulfillment service, which in turn generates events for the email automation service to notify the customer of the product status. The email automation service may *respond* to the product fulfillment service by recording a link click from the customer, and notifying the product fulfillment component that the customer is ready to receive their product.

5.1.3 Stream Processing

Stream processing is the practice of taking action on a series of data (events) at the time the data is created [9]. A data stream represents an infinite, unbounded dataset [4]; unlike batch processing where data sets are finite, data streams are ever-growing. Data streams are sequential and generated at high speed; typical examples of systems outputting a data stream are financial markets, sensors, application logs, monitoring systems, and more [10]. Data streams are characterized by three key attributes: 1) Ordering—data streams are sequential; 2) Immutability—once an event is generated, it is not editable; in case the system's state is changed, a new event is recorded; 3) Ability to replay—data streams can be replayed to perform new procedures [4]. Unlike the request-response paradigm, data stream programs are non-blocking; and unlike batch processing, stream processing has much lower latency [4], and is often employed in real-time computing [9]. Furthermore, stream processing is tied to several concepts which affect how it manipulates incoming events: 1) Time—there are multiple timestamps connected with each Event, including the actual time of the event, the time it was produced to the stream, the time it was processed, etc. It is important to note that time recordings may be captured in different time zones, which could result in inconsistency across the system; 2) State—most stream computing platforms need to record the interim app state; for example, a

simple moving average calculator needs to record the last SMA value and the number of entries [11]. As a result, the stream processor needs access to permanent storage in order to be able to recover from crashing without losing data. The app state may be internal (in-memory), or external (saved to a database); 3) Stream-table duality—relational databases normally only contain the latest state of the world, while a stream needs to be fully saved in order to retain the ability of replaying; 4) Time window—there are multiple considerations to take into account when conducting time window-based computations, such as how long the window should be, how frequently the window "slides", until when can events be added to their window, and so forth. [4]

Design patterns. The design of a stream processing software depends on its business logic; some processors require extensive use of permanent state, some employ sub-streams, while others are more "lightweight" and conduct simple computations. Single-event processing is a stream processing design pattern which conducts filtering and mapping on individual events. This is the simplest architecture to recover from in the event of a failure, and will be the inspiration for our platform. On the other hand, stream processing with local state requires keeping the application state in local or persistent memory; a simple moving average application may be built with this pattern. Multiphase processing (repartitioning) involves building a multistep application; each step relies on the computational results of the app "before" it. For instance, any app that conducts aggregation over multiple streams, where one stream is handled by one app instance, will need a "second step" for computing the actual aggregated result. Stream-table joining is a stream processing pattern in which one stream processor requires access to a database; in order to prevent latency issues or overloading the database, a local cache is kept, which is updated by a different stream to ensure data isn't stale. In a streaming join pattern, multiple streams are merged into one based on event metadata and the time of collection. Finally, regardless of the design pattern utilized, stream processing software has to deal with out-of-sequence events. These messages were not sent in the intended order, but still have to be processed just like the rest of the stream. The program has to define the time window within which it accepts out-of-sequence events, and update the results accordingly, which can be tricky depending on the business logic. [4]

5.2 Apache Kafka

"Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications" [12]. It works as a distributed event-streaming service with persistent storage and low-latency data pipelining; similarities can be pointed out between Kafka and ETL software, data integration tools and big data systems like Hadoop. [4] The platform is used by more than 80% of Fortune 100 companies, including Netflix, LinkedIn, Airbnb, Oracle, and others [13], which is why Kafka is considered the industry standard for event-driven data pipelines. And not only that, Kafka is the perfect choice for our platform as users will expect their data to be delivered consistently and in a timely order, which is what Kafka's architecture is built for.

Today's state-of-the-art big data systems are considered: 1) reliable; 2) scalable, and 3) maintainable [4]. Reliability will ensure that systems communicating through our data pipeline

platform are guaranteed to send and receive events securely, promptly, and without data corruption. Due to the nature of our problem, we can expect to be connecting tens to thousands of different servers, so scalability will play a crucial role. Finally, our platform must be maintainable, especially in terms of future updates from the development team—as the APIs of systems relying on our data pipeline evolve, the team will need to maintain the specific components of our platform connected with those APIs in an easy manner, with results guaranteed by unit tests.

Kafka fulfills all three aforementioned requirements thanks to its robust, distributed architecture. The platform incorporates permanent storage of received events in a commit-log-alike structure, which enables us to achieve a consistent system state across all receiving applications (Consumers), even in case of failure.

5.2.1 General Data Flow

The data flow in Kafka incorporates three key actors: the **Producers,** the **Kafka Cluster** and the **Consumers (Consumer Groups).** The Kafka Cluster represents a cluster of servers which Kafka is deployed to (the streaming platform itself). Producers and Consumers are external systems connected to the Cluster with the goal of sending and reading/processing the event data, respectively. Instances of Consumers are grouped into **Consumer Groups** (except in special cases.) When sending data, Producers write Messages to **Partitions** of a certain **Topic.** Topics represent conduits for transferring Messages from Producers to Consumers. This process is conducted by **Brokers,** which fetch data from Partitions of Topics, and distribute the event data (Message) to the respective Consumers of the subscribed Consumer Group.

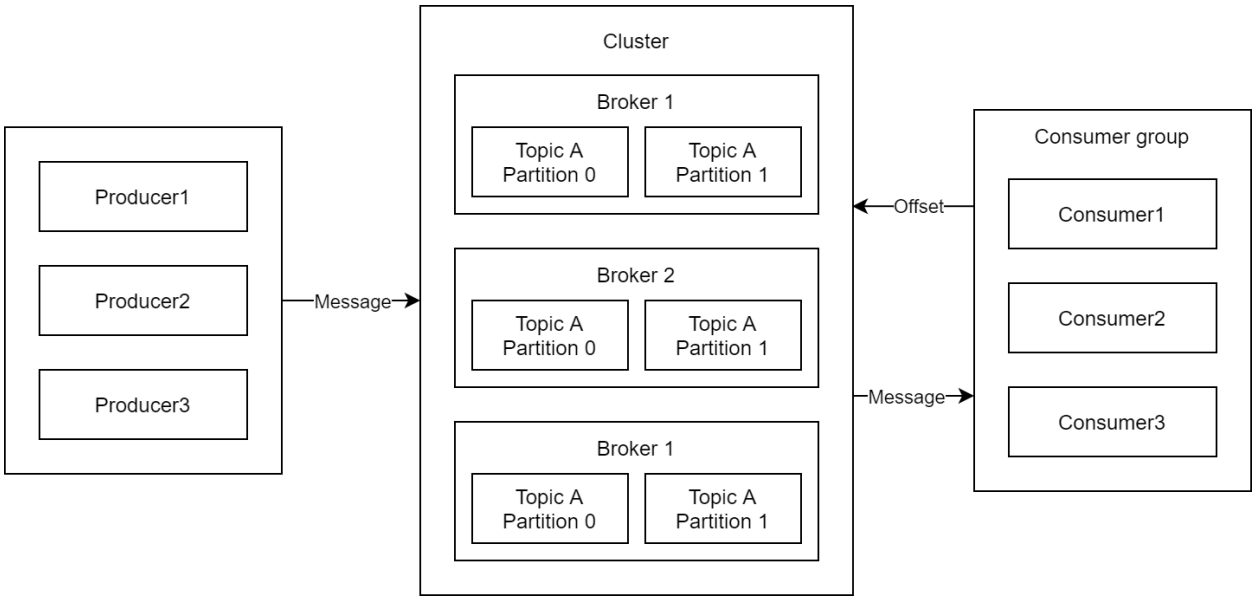

Figure 5.1: Kafka cluster architecture

5.2.2 Decoupling

Apache Kafka enables the creation of a data pipeline between sources and consumers of data, even if the sending/receiving systems are heterogeneous, through decoupling. The only overlap between systems exists in the message schema, for which Apache Avro provides an effective solution. Producers and sinks are also decoupled time-wise—thanks to the commit-log type of message storing, Consumers need not remain in sync with the Kafka cluster. In case a Consumer fails, it can simply re-read the commit log from the last read message onward.

Schema. Producers and Consumers are only coupled by the schema of the Message content. As applications evolve and schemas change, this can become a problem—if the Producer's schema changes, the Consumer must have a way to either use the new, updated schema automatically or have forwards compatibility towards the updated Producer's schema. Likewise, an update can happen in Consumer's internal procedures to use a new schema, so the program must retain backwards compatibility in order to read the Producer's messages. An example of this situation is the rolling upgrade: updating the Producer's server must not break the Consumers, and vice versa. There are multiple ways to tackle this issue. **Apache Avro** is designed with schema evolution in mind and is one of the preferred tools for this problem. Avro is a data serialization framework [14] used in many distributed systems (such as Kafka and Hadoop) to standardize data types across systems. Avro saves data as an Avro Object Container File, which contains a header and data blocks. The header defines the file metadata, including the schema definition [14]. Avro has built-in support for schema evolution, which makes it easy to update the schema on either end of the system without conflicts. A **schema registry** can be used with Avro to define a central server which hosts the most up-to-date version of event schema, for all programs to use. The identifier for the schema is stored as part of the Producer's record, which is then used to identify the schema in the registry upon decoding by the Consumer. [4]

5.2.3 Architecture

Kafka is typically deployed as a cluster (the Kafka Cluster). A single node in the cluster is called a **Broker;** one controller is assigned the role of **cluster controller,** and is auto-elected. Brokers host topics by receiving messages and allowing Consumer Groups to read them. A Topic can consist of multiple **Partitions;** there can be an arbitrary number of Partitions, configurable per Topic. Typically, a Topic is configured to have the number of Partitions equal to the number of Brokers in the cluster (or a multiple of that number) [4]. (The number of Partitions requires estimation of multiple parameters—such as expected throughput, available system resources, etc. [4].) That way, each Broker holds at least one replica of the Partition, which provides extra data redundancy in case of failure. Each Partition is assigned a **leading Broker,** wherein only that Broker may accept requests from Producers and Consumers regarding that the respective Partition—all the other Brokers' replicas are hosted purely for fault-tolerance purposes. This distributed architecture of replication is what provides Kafka its high redundancy.

Partitions. Partitions are structured much like an append-only commit log. A message represents a "new line" in the Partition, containing the Message key and its contents. Each partition is assigned an ID, and Producers may choose which specific Partition to write the message to, or may leave that decision to the Cluster. The ID of the partition to write to is

determined by applying modulo function to the hash of the key passed, which ensures that providing the same ID always writes to the same Partition. Producers normally send Messages to Brokers in batches to improve performance. When reading from the Partition, the Consumer keeps track of the offset of the last read Message, reporting it back to the Cluster configuration server (ZooKeeper). In order to guarantee Message reception by Consumers, data on Partitions can be retained for a configurable amount of time (usually up to 7 days) or until a size limit is reached (until 1GB). Message size is also configurable and must be synchronized on both Producer and Consumer clients, as failing to do so may result in Consumer's inability to read large Messages. **Failed leaders.** In case the leader of a Partition fails, the cluster controller has to elect a new Partition leader. The next Broker in the list of Partition replicas is chosen for this role, with all the relevant nodes being alerted of the change. **Failed consumers.** Kafka Consumers send heartbeats upon reading a Message to their designated group coordinator broker, thus signaling an active, well-functioning state. However, in the event that the Consumer goes down, the group coordinator will consider it "dead" and will trigger a **rebalance** of partitions. This means that ownership of Partitions within the Consumer Group will have to be re-assigned, which implies downtime for the entire Consumer Group. [4] It is important to consider this case in the context of the production environment, as it results in several scenarios that have to be handled: 1) fault-tolerance in case of the entire Consumer Group unavailability; 2) data reliability (particularly, preserving historic order of events) in case of rebalance; 3) fault-tolerance for the Consumer that failed itself.

Replication. As previously stated, Kafka uses Topics for categorizing Messages, and each Topic may contain multiple Partitions. The replication factor determines how many replicas are stored in the Cluster, and is typically a multiple of the number of Brokers. The Partition leader is the only node which can accept read and write requests from the Producer, with all the other replicas simply "mirroring" the changes; however, the Partition leader's job is also to ensure follower replicas are kept in sync with the leader. This is done via a similar mechanism that Consumers use to read Messages; followers send `Fetch` requests to the leader requesting the next Message. The requested Message is determined by the offset, and is incremented sequentially (in other words, the follower will not request the message with the offset 3 before writing messages 1 and 2.) In case a replica fails to fully sync-up its commit log within 10 seconds, it is considered "out-of-sync" and can not be declared leader in case the old leader fails. [4]

Storage and segmentation. Partitions in Kafka can not be split between Brokers or disks (although the mount point may consist of multiple disks in case of RAID configuration); thus, the size of the Partition is limited by the size of available storage. This may present a potential problem in case some Brokers have significantly larger storage available than others. When creating Partitions, Kafka assigns them to Brokers in a round-robin manner, ensuring even distribution; rack configuration is also factored in starting with version 0.10.0. The Partitions are then added to the directory containing the fewest Partitions on that node, ensuring even distribution within the Broker's storage system. **Segments.** Partitions are split into **segments** which allow for easier search and purging of Messages. A segment is retained as per the configuration parameter—for example, up to 7 days or up to 1GB in size. The segment that is currently in use by Kafka for writing is called the **active segment** and can not be deleted. **Message format.** Each segment is stored as a single file, and messages stored inside the segment

use the same format as that received from the Producer, and later sent to the Consumer, which results in very little resources needed to process the Message upon writing. Each Message contains a key, value, offset, size, checksum and compression. In case of compressed Messages, the Broker will receive a batch of compressed Messages wrapped into a single "wrapper messages", which is then forwarded to the Consumer and "unpacked" on that end. This also means that sending compressed Messages saves storage space on the Broker, in addition to lowering network load. **Indexing.** In order to quickly fetch Messages at the requested offset, Kafka keeps an index for each partition, mapping offsets to file segments and their positions in the segment files. Indexes, too, are broken into segments; in case of index corruption, segments are reread to reconstruct the index. **Compaction.** In order to keep data relevant and avoid consuming all available memory, Kafka performs compaction on segment files. There are multiple modes of compaction—some applications may require persisting data up to a certain size or for a certain amount of time, while others may care about the most recent state of the system, discarding previous entries. Compaction is usually executed on a Topic at 50% "dirty" records (Messages received after last compaction). [4]

The Controller. Each Cluster has an auto-elected controller node in charge of electing Partition Leaders. There is always at most one Cluster controller; upon entering the Cluster, the new Broker will attempt to register itself as the controller, and will receive an error if there already exists one. In that case, the Broker will create a ZooKeeper watch, ensuring that it gets notified in case the controller fails. In the event of a controller failure, all watching Brokers will try to elect themselves controller, but only one will succeed, with the rest receiving error messages. The **epoch number** of the controller is incremented with every new controller election, and ensures that Brokers can safely ignore old controller's instructions by checking the epoch number of the command. [4]

Request processing. Communication between Kafka clients and the Cluster is done via a binary protocol on top of TCP, that uses request-response message pairs as APIs [15]. Furthermore, "all requests sent to the broker from a specific client will be processed in the order in which they were received—this guarantee is what allows Kafka to behave as a message queue and provide ordering guarantees on the messages it stores." [4] The standardized binary format defines a header with the following fields: *Request type* (API key); *Request version; Correlation ID* (unique identifier of the request); *Client ID.* [4] Kafka uses an **acceptor thread** for creating connections, which are handed over to **processor** threads for handling; requests from clients are then placed in a **request queue** to be picked up by **IO threads.** Responses are placed in the **response queue** to be picked up by the network thread for sending the response. There are three common types of requests that Kafka clients may send: **1) Produce requests** for writing Messages; **2) Fetch requests** for reading Messages, and **3) Metadata requests** which provide information on the Topic of the client's interest. Metadata requests may be sent to any Broker, but the former two may only be sent to the Partition leader. Sending a produce or fetch requests to non-leaders will result in an error, after which the client will have to refresh its data and retry the request. A produce error may also occur in the case that the client has insufficient permissions for writing to the partition; if the `acks` parameter is misconfigured; or if there are not enough in-sync replicas to successfully fulfill the `acks` value. On the other hand, a fetch error may occur if the client is requesting a message that is too old and has been deleted (too low offset), or requesting a non-existent message (too high offset). A unique aspect of Kafka is that

messages are sent directly from the file (or its cached version) instead of using an intermediate buffer, which provides a significant performance improvement. Furthermore, clients may configure their request to receive data within certain limits; for example, the client may request *not* to receive a response until Kafka can send over 10KB worth of data, or there may be an upper limit to the amount of memory the client can handle; likewise, there may be a timeout defined by which the Cluster must send a response. It is also important to note that only Messages that are present on *all* replicas may be read by the client. This is because if the leader were to fail and other replicas did not contain the message, the client would get inconsistent results. For this reason, only fully-synced Messages may be read. [4]

Multiple Clusters. Large cloud applications often employ more than one cluster, or even multiple datacenters for increased availability, redundancy, and even to comply with legal regulations. There are three common architectures used to set up a multicluster environment: **Hub-and-Spokes, Active-Active** and **Active-Standby.** The Hub-and-Spokes architecture considers one cluster as the "central" for the network, with the other clusters reporting back to it. This type of architecture may be used in case the other clusters are certain not to need any remote data or procedures, as this may break the decoupled relationship of the clusters and introduce unnecessary complexity. Alternatively, a leader-follower architecture can be applied to the example of having one central cluster, and a backup cluster for reports. The Active-Active architecture utilizes multiple "leader" clusters, and is the most scalable, resilient, flexible, and cost-effective option [4], as it provides full functionality, and data redundancy across the entire network of clusters. However, the Active-Active architecture comes with the drawbacks of every multileader distributed application: the challenge of handling "read your own writes" mechanism, monotonic reads, consistent prefix reads, conflict resolution, and so forth. Additionally, the mirroring process has to be carefully handled so as to avoid an endless loop—this can be achieved by creating namespaces for Topics. The Active-Standby architecture is normally used as a safety net in case the entire cluster breaks down. This is a simple setup where all events from the main cluster are simply mirrored into the backup cluster, without any conflict resolution protocols. Alternatively, it is possible to set up a Stretch Cluster [4] which includes only one Cluster, but whose nodes are scattered across three or more data centers for increased resilience; this solution does not require any mirroring as all Brokers are part of a single cluster, however, network latency may pose a performance issue. **MirrorMaker.** Apache Kafka ships with MirrorMaker, a tool for mirroring multiple data centers. In its essence, MirrorMaker is a collection of Consumers coupled with a single Producer; the Consumers listen to incoming messages from the source cluster and pipe them into the Producer, which relays the events into the target cluster. Messages are committed to the original server which ensures no data loss. MirrorMaker can be run on either the source or the destination system; normally, it is better to run MirrorMaker on the destination system, but this is not the case if SSL encryption is used as it puts more performance pressure on the Consumer than the Producer. Because our multiple data centers will store potentially sensitive data, it will have to be encrypted in the pipeline, which means our MirrorMaker instances will be executed in the source cluster. **Failover.** A growing number of organizations are employing preventive failovers and practicing "planned disasters" with tools such as Netflix Chaos Monkey [16]. Planned, controlled failovers of Kafka clusters can be achieved with no data loss if the mirroring process is allowed to complete copying of the data. However, in production environments, this is rarely the case,

which means that a failover will almost always result in some degree of loss of data. There is no perfect solution to handling a failover; the clients will either start reading from an offset that does not exist, or will have to go back in time and process already-seen messages as duplicates. Another issue arising from a failover is what to do with the failed cluster—the recommended solution is to clean the old cluster of its data and start mirroring from the new leading cluster. This way, while some of the old cluster's data will be lost, there will be no historical inconsistencies. [4]

ZooKeeper. A Kafka Cluster requires assistance in handling metadata, such as Consumer read offsets. **Apache ZooKeeper** is typically employed for this role. ZooKeeper is a distributed service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [17]. This top-level Apache project [18] was developed to simplify the storage and retrieval of configuration data for distributed applications. Like many systems that rely on it, ZooKeeper itself is a distributed, scalable, fault-tolerant server, typically deployed in a cluster called an **ensemble.** The registration of Kafka nodes is conducted through ZooKeeper. Each Broker registers an ephemeral node—a session-temporary node holding important metadata about the Broker [19]. Each Broker is also assigned an ID, and no two Brokers in the Cluster may have the same ID. In case of a Broker being removed from the Cluster, all data structure information regarding it (e.g. list of replicas of a Partition) will retain its ID; that way, a replacement Broker can be inserted "in-place" with the same ID, thus picking up the same Partitions and Topics of the old Broker. [4]

5.2.4 Reliability

Reliability is discussed extensively in this article. Apache Kafka supports different "levels" of reliability; certain systems may allow for data loss (e.g. tracking user interaction), while others (e.g. financial transactions) put emphasis on reliability more than any other aspect of the system. Kafka's extensive configuration enables setting up the optimal "level" of reliability in the Cluster. [4] Our platform, being a proof-of-concept solution, will not aim for 100% reliability; creating a generalized system for *every* use case, from processing email open notifications to transaction data, would be extremely difficult, as the vast array of systems relying on the platform will all require a different level of reliability. Thus, we will aim for good reliability, but will not make perfect reliability guarantees. The trade-offs in reliability will also give our system better performance in terms of availability and performance. [4] The following parameters in Kafka allow for configuring the reliability of our Cluster:

Replication factor. The replication factor controls how many copies of the Partition are kept in the system, and may be configured as a default at the Broker level. This parameter affects the system in multiple ways—high replication factor means a more fault-tolerant Cluster, but also the need to have at least N brokers up and running. More replication also means more latency in awaiting the Cluster response in case of `acks=all` produce requests, as well as more storage required to host the data. [4]

Unclean leader election. This scenario occurs when the Broker hosting the leader replica fails, and there are no in-sync Partitions available to promote as leader; for example, if the other Brokers have, too, failed, or were not synced in time due to network lag. Kafka allows for the promotion of an "unclean" leader (an out-of-sync Partition) if this setting is enabled, but this may

not be the desired behavior in systems requiring utter data accuracy. This is because electing an unclean leader means the system is practically guaranteed to encounter inconsistency (or loss) of Messages. However, some systems may prefer having high availability even at the price of small inconsistencies in the data stream. [4] In an ideal situation, we would be able to configure this setting on a *per-Topic level;* however, this granularity is not allowed, therefore our platform will enable unclean leader election for the entire Cluster.

Producer. No matter what degree of reliability is configured on the Broker, the Producer must still configure its own reliability-related parameters separately, as well as implement the right algorithm for dealing with possible errors. One of the most important values determined by the Producer is the `acks` parameter, with the value `0` giving high throughput but no guarantees, and `all` consuming more resources but ensuring a high degree of reliability. Furthermore, in the case of errors, the Producer may need to (keep) retrying, as *retriable* errors can usually be solved by resending the Message. An additional issue lies in the nature of networking—if the Broker has received and committed the Message successfully, but its successful response fails to reach the Producer, the latter may retry sending the Message, therefore committing the same event twice. This is why Kafka does not offer the guarantee of *unique* Message, but that Messages will be committed *at least* once; a possible solution lies in including a unique identifier in the Message, or sending **idempotent** Messages (e.g. "User has 30 coupon points" vs. "Add 20 coupon points to User"). There are also other errors that may occur—called **nonretriable** errors, that require extra handling from the Producer's end. [4]

Consumer. As the sink of the system, the Consumer plays an important role in building a reliable pipeline. Several design choices dictate what degree of reliability is built into the Consumer: 1) The Consumer may be part of a larger Consumer Group, or it may subscribe to all Partitions directly. This may represent an SPOF (Single Point of Failure) of the system in case no other Consumers are processing the aforementioned Topic; 2) With the offset reset configuration, the Consumer may read all Messages from the beginning, leading to possible data duplication, or it may read Messages from the end of the Partition, resulting in potential data loss; 3) Committing frequency: the Consumer may opt for auto-committing which provides good reliability but no solid guarantees, whereas custom commit control may offer greater precision at the cost of higher complexity. Furthermore, some use cases may require Consumers to buffer received Messages to preserve order of the queue. This may happen when the time needed to receive new Messages is shorter than what it takes to process them. Another possible issue faced by Consumers is that of "exactly-once" semantics; some programs need a guarantee that all processed Messages will produce a unique, non-duplicated result. This can be achieved using a key-value store, where Messages are identified using either the combination of Topic-Partition-Offset as a complex key, or, more commonly, through a unique identifier passed inside the Message contents. Finally, in the case of the Consumer needing to store the state of the computation, it may be possible to create a special Topic for committing interim results. [4]

5.2.5 Performance

Considering Kafka's role as a pipeline platform, a substantial amount of time can be devoted to tuning the parameters of the Cluster for maximum performance. Because of the app's distributed

architecture, a small performance optimization replicates manyfold across the cluster, and can result in great time, memory, or cost savings.

Broker count. The number of Brokers in a Cluster impacts many aspects of Kafka's activity—from replication to performance. For example, fewer Brokers may result in lower costs, yet lower reliability; more Brokers offer better redundancy but may result in increased latency, especially if the replication factor is configured to deploy at least 1 replica to each Broker.

Storage. Starting with the host machine itself, hardware plays a key role in Broker performance. Reading/writing throughput is dependent on the machine's disk; SSDs normally outperform HDDs in this criteria. It is also important to factor in the type of connector used for attaching the storage unit (e.g. SATA); HDDs may also be configured in RAID structure. Furthermore, disk selection is important in terms of storage size; here, HDDs outperform SSDs in terms of capacity per unit of cost. In addition, different filesystems provide different read/write performance. Here, XFS tends to beat EXT4 out of the box, and is becoming the default filesystem on a growing number of Linux distributions.

Memory. Kafka makes use of caching to improve read/write speed. The more available memory Kafka has access to, the more log segments can be cached, further increasing performance. For this reason, it is recommended that Kafka is not run in parallel with any other memory-expensive application, as it may limit Kafka's caching effectiveness.

Networking. The key networking configuration parameters are throughput limitations—bandwidth, buffer sizes, etc. The more relaxed limitations imposed on Kafka, the greater the performance.

ZooKeeper. Kafka is most commonly deployed with ZooKeeper for metadata handling. Normally, Brokers communicate with ZooKeeper only in case of changes to the system; the biggest burden on this system comes from Consumers. If Consumers choose ZooKeeper for storing Partition read offset, this may result in a large amount of traffic towards ZooKeeper. It is thus recommended that Consumers use Kafka instead of ZooKeeper for storing read offsets. [4]

5.2.6 Maintenance and Monitoring

In a complex system with multiple Clusters and dozens of Brokers within each Cluster, administration becomes an important issue. Kafka provides a suite of CLI tools used for connecting to the Cluster and managing settings. These tools range from topic operations, over replication, general configuration, Consumer Groups, to clients, and even include more dangerous operations such as changing the Cluster controller or removing Topics. [4] While these tools will come in handy on production, we will also aim to automate as much of the setup and administration work as possible through our CI/CD routine. Neha et al. [4] recommend tools such as Chef or Puppet for maintaining consistent configuration settings across different host machines.

When it comes to monitoring, Kafka offers a host of measurements reporting the state of operations. They are exposed via the Java Management Extensions (JMX) interface; details of the JMX port are stored in ZooKeeper, and can be used for a collection agent to fetch the accumulated data. These metrics represent Kafka's internal observations, and should be coupled with an external metrics server (for example, to check the availability of Brokers) that does not

depend on Kafka [4]. Neha et al. [4] define the following metrics as essential: 1) Under-replicated partitions: if there are partitions not caught-up, this signals to a wide array of possible problems, from resource unavailability to Broker outage, to a network connection issue; 2) Broker metrics, such as active controller count, request handler idle ratio, all topics bytes/messages in/out, leader count, offline partitions, request metrics (eight metrics each); 3) Topic and Partition metrics, which may be useful to clients; 4) JVM metrics—particularly, monitoring garbage collection; 5) OS metrics, such as memory usage, CPU load, disk IO, network usage, etc. Kafka's official Java client libraries also come with built-in metrics that should be captured for analysis. [4] Many of the provided metrics have to be monitored on a constant basis—alerts should be set up in order to capture outlying conditions. It should also be noted that many measurements require conducting tests on the cluster setup in order to establish baseline values, such as the average request latency; alerts may be set up at a certain threshold above the baseline. Furthermore, Kafka supports throttling in order to prevent highly-demanding clients from overwhelming the system.

5.3 Spark

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters [20]. Spark was inspired by Hadoop; specifically, by Hadoop's heavy reliance on disk operations. Spark uses in-memory processing, with result caching in between operations, thus having much faster execution. Spark is commonly used for interactive queries, real-time analytics, machine learning, graph processing, and more. The Spark framework consists of multiple components, with Spark Core being the essential platform, and Spark SQL, Spark Streaming and Spark GraphX available for specific use cases. [21]

Spark Core. Spark Core is the foundational component of the Spark framework, providing a general execution engine with in-memory capabilities, an execution model, and different APIs for ease of development [22]. Spark's distributed data processing ability is built into Spark core, as well as its fault recovery mechanisms, scheduling and management of jobs on the cluster, and storage system interaction [23].

Resilient Distributed Datasets (RDDs) are at the core of Spark's programming paradigm. RDDs represent fault-tolerant read-only objects, which support two main operations: **Transformations,** that result in new (immutable) RDDs, and **Actions,** the results of which are passed back to the driver [23]. Because RDDs are split into logical partitions, operations performed on an RDD may be done in parallel by different nodes in the cluster. This also means that in case of failure, an executor node containing a replica of the partition can take over the process and complete the computation.

Architecture. Spark is a distributed programming framework, consisting of the **Driver Program,** the **Cluster Manager** and the **Worker Nodes** [24]. The Driver represents the program utilizing Spark for computing; the Driver calls `SparkContext` to connect to the Cluster Manager (Standalone/YARN/Mesos/Kubernetes), and thus acquires access to **executors** on Worker Nodes, which then receive the application code and **tasks** for processing [24].

Libraries. Spark provides multiple libraries on top of the Spark Core component for specific use cases. The **MLib** library enables data scientists to create machine learning

algorithms at scale, with rapid, interactive in-memory processing. **Spark SQL** is a distributed query engine used for fast, interactive queries on large datasets. Spark SQL relies on **DataFrames,** a data abstraction that includes support for structured and semi-structured data [25]. It is used by business analysts and developers alike. **GraphX** provides ETL and computation operations on distributed graphs. However, it is not used as a graph database as GraphX graphs, like RDDs, are immutable. Finally, **Spark Streaming** is a library for real-time data processing, which performs RDD transformations on mini-batches of data. This allows developers to use the same code for real-time stream operations as for batch processing, and lets organizations analyze data "as it flows in" instead of running long, complex batch jobs recurrently. Our platform will use Spark Streaming for further transformation of event data, and user-defined batch processing before forwarding into Consumer applications.

5.4 Docker

Docker is an open platform for deploying isolated software environments, called containers [26]. Docker is one of the most widely-used container tools, being present in 44% of enterprises in 2022 [27]. Having a software environment is required at every step of the development process, from coding to production. However, as devices used by an organization tend to vary in system properties, their respective environments usually differ. These differences can lead to unexpected bugs in application behavior, compatibility issues, increased difficulty in developing the application code, etc. Docker effectively solves this issue by creating a virtualized, "loosely-isolated" environment [26] for deploying applications. Containers enable developers to create, debug and deploy software in a uniform environment, no matter which machine the code is being executed on, or which stage of the development process is taking place. The uniformity of containers yields many benefits: 1) an application ran in a container should always produce the same results (assuming same input/parameters), no matter which device the container is hosted on; 2) due to the nature of containers' "loosely-isolated" environment, the risk of interference with application processes or unexpected behavior is minimized; 3) developing the application in the same environment as the production host removes most, if not all, differences from the developer's local system, leading to a facilitated development process and more predictable application behavior; 4) as containers can be custom-tailored to the developer's exact needs, it is possible to create extremely lightweight, efficient environments, allowing teams to run multiple containers on the same host device; 5) running the application in a virtualized environment creates an additional layer of isolation, improving the security of the environment; many container images available on Docker Hub [28] are also audited, further strengthening the security of the deployment.

5.4.1 Virtualization and Containerization

"Virtualization is a technology that creates a abstracted layer over computer hardware that allows the hardware of a single computer to be utilized and divided into multiple virtual computers, known as virtual machines." [29] Containerization is a more lightweight form of virtualization, wherein the container does not create a complete guest OS, but relies on the container engine and the host kernel for execution. Containerization in Docker is conducted via the Docker Engine,

which uses a client-server architecture [30]. The long-running daemon `dockerd` process runs the Engine, while a set of APIs provide functionality to the system via the `docker` client CLI. Like other virtualized environments, with the right configuration, a container may have access to permanent storage, networking features, input/output devices, and more. However, as containerized environments run "on top of" the guest operating system, they make less of an impact on the performance of the machine and do not provide complete isolation from the system [31]. Containers may also be configured not to provide all the services of a complete OS for extra performance gains. All of this is what makes containers a more efficient tool—CPU performance-wise—in comparison to traditional Hyper-V [29]; as a result, containers are often used to deploy microservices [29] to the cloud, and even to run multiple containerized apps on the same machine; they are frequently employed in the development/testing process because of their ease-of-use/simple set-up compared to Hyper-V [29] and lower resource requirements. We will use containers to create our proof-of-concept; however, the final commercial project should try to avoid containers as they provide worse performance than using the actual host machine, and have been noted to provide worse networking performance than Hyper-V virtualization [29].

5.4.2 Build Process

Docker containers are built from **container images,** a concept similar to system images [32]. For example, "ubuntu" is a Docker official image [33] built from official Ubuntu tarball distributions by Canonical. However, unlike traditional system images, container images provide greater flexibility. Container images can be modified with a Dockerfile [34], which specifies the system image to use when building the container, as well as the operations to perform on it in order to set up the environment required to run the application. These instructions range from copying a directory in the application repository (e.g. the folder with app source code) to executing shell commands (e.g. compiling app code and running the executable.) During the build process, Docker executes instructions from the Dockerfile sequentially, fetching the image and processing the instructions. It is also possible to create multiple Dockerfiles for the same application repository, depending on the organization's needs. This process allows replication of the desired environment across virtually any machine running Docker, regardless of its host operating system and exact hardware specifications.

Configurable Dockerfile and CI/CD. A unique aspect of Dockerfiles is their syntax, which resembles that of a Batch file[1]. The declarative Dockerfile language enables developers and DevOps teams to create ready environments across any number of machines by changing the Dockerfile code, instead of applying the changes to the host machines manually. The available instructions allow for a range of possibilities: running commands, setting environment variables, exposing ports, conducting health checks, setting user configuration, creating the entry point for the container (application), and even setting variable arguments for the Dockerfile itself, plus many more commands [35].

[1] Theoretically speaking, the Dockerfile language can be considered Turing-complete if it can access an executable in the container which can process a Turing-complete programming language (in other words, the command "RUN python3 <code>" would arguably render Dockerfile a Turing-complete language.) However, Dockerfile syntax is *not* widely regarded as a programming language, but rather a declarative syntax used strictly for configuration.

Docker also enables developers to export container images into the organization's registry, thus compressing the need for extensive Dockerfile configuration even further, by simply "pulling" the image and executing it on the target machine [36]. This platform also proves a valuable tool for CI/CD; using tools such as GitHub Actions [37], it is possible to fully automate the updating of Docker containers, without manually accessing the server on each deployment.

6 Solution and Documentation

Our solution for building a “data pipeline platform” is to create a high-availability, high-reliability cluster of servers for fetching or accepting data from selected source systems (deployed via an event-driven microservices architecture), routing them through one or more data centers with built-in redundancy, transforming the data from the source system’s schema into the destination system’s schema, complete with any aggregations/computations defined by the user, and finally sending the data into the destination platform with as high reliability as possible, but no explicit delivery guarantee. All of this has to be conducted in a controlled, secure environment; where only the authorized systems may communicate with the components of our platform; credentials required for connecting to source/endpoint systems are secured; and the data passing through our platform is encrypted, inaccessible to 3rd party systems unless directly sent. Furthermore, our platform has to support both the semantics of “at-least-once” and “exactly-once” message delivery, so as to enable future integration with a wide array of 3rd party systems. As previously stated, the platform shall offer a high degree of reliability, executed through a proprietary algorithm for sending the message to the endpoint, but will not provide guarantees, and therefore will not be the ideal solution in case avoiding data loss or duplication is one of the user’s crucial concerns.

Because it is our goal to enable non-technical users to create data pipelines, a fundamental goal of our platform is to provide an easy-to-use, graphical interface for mapping source systems to endpoints, with optional data aggregation and transformation as the data passes through our platform. The supported source/endpoints shall be the APIs our platform is integrated with; meaning, there exists a connector for fetching data from the source, and an adaptor for aggregating/transforming the events into the schema of the endpoint.

6.1 Architecture

Our data pipeline platform will consist of several independent microservices, communicating in an event-driven, request-response manner.

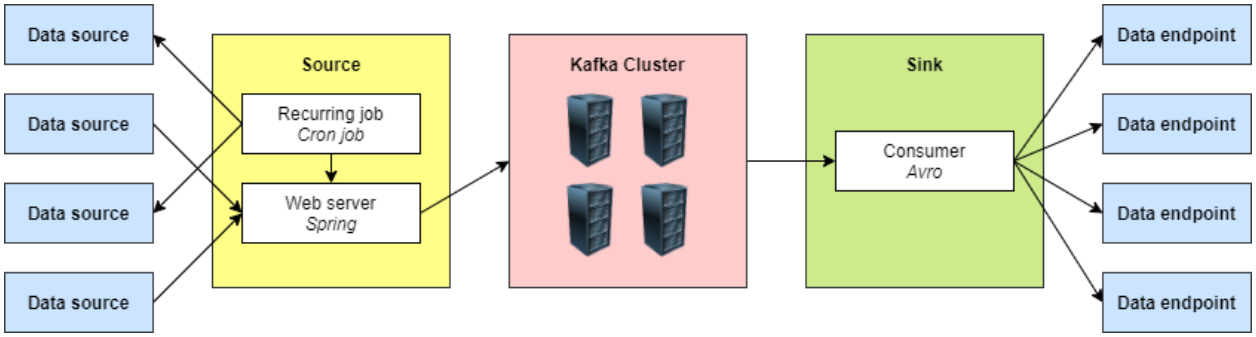

Illustration 6.1: Platform architecture

6.1.1 Source

This component represents the source of events flowing into our data pipeline. Events may be generated by one of two methods: 1) via webhooks/notifications sent from a 3rd party API to our system, or 2) by querying the selected 3rd party system on our own, using a recurring job. This component outputs event data for further forwarding into the Kafka Cluster.

Webhooks. In case the source system sends data via a notification/webhook (e.g. Stripe sending a payment notification), the webhook must be registered. This means we will need a high-availability web server for accepting requests; this component will be bundled into the Kafka Producer Client.

Querying. Certain source systems do not offer webhook integration, so these have to be queried "manually" via an API request. There are many frameworks for creating and running recurring tasks, however, since we'll be deploying our platform to Linux servers, we can build a simple Cron job that runs a script for fetching data.

Because the platform aims for high redundancy and reliability, this component will not be required to transform the received event data in any way; it will merely "forward" the data into the Cluster as-is, in order to preserve its integrity. When it comes to other requirements of this system, the web servers used for receiving notifications must provide high availability in terms of uptime and processing power, as some source systems may not resend data in case we fail to respond. Furthermore, we may have to buffer the data before sending it into the Cluster, in case the Cluster fails to commit the message. Regarding API query jobs, there shall have to be a monitoring/management process for overseeing the individual query processes, and ensuring they are timely executed. Events received via notifications cannot be planned in advance, however, it is important to consider how frequently our Cron jobs will run for each of the source systems. Most APIs impose some limit on the number of requests from a client per minute/hour. The frequency and deployment of query jobs has to be carefully planned as spawning a large number of processes may pose scalability issues and loss or interruptions in the data stream.

Therefore, it is best to deploy the webhook servers and query worker manager nodes as different microservices, on different nodes.

6.1.2 Kafka Cluster

Once the Source component(s) generate event data, the message will be transmitted through Kafka to the Sinks. In a commercial project, our Kafka network would rely on multiple Clusters, connected in an active-active relationship. An example setup would entail one Cluster for every major region, such as the US, Canada, South America, Europe, Asia, Australia & Oceania, etc. Ideally, each Source system would send data to the nearest cluster, which would then mirror its data onto other clusters with Kafka's MirrorMaker mechanism. Furthermore, in order to achieve the best performance possible, Kafka would be installed directly on the host machines, without virtualization or containers. However, for research purposes, we have opted to use Docker; therefore, a containerized installation of Kafka, for easier setup and development. This allows for faster deployment and testing, but degrades the performance of the system. Additional optimization would also be performed on the host OS itself, in order to allow for greater caching capacity and higher throughput to Kafka client software.

In order to connect to the Cluster, our architecture will include a **Kafka Producer Client** component, configured to match the nature of the event data to-be-sent. Some messages may require high throughput, low latency and may tolerate data loss, while others rely on data delivery guarantees. For simplicity's sake, we will organize our data in the Kafka Cluster so that *one source equals one Producer Client, equals one Topic.* Therefore, there will be no mixing of event data in different Topics. On the other end of the system, a set of **Kafka Consumer Clients** will be deployed to fetch events and send them towards the endpoints—Sinks. Consumers will be deployed as microservice instances organized into Consumer Groups, with each Consumer Group subscribed to only one Topic.

6.1.3 Sink

Finally, once we've received the data from the Broker, we will use **Apache Spark** to extract, load and transform (ETL) the data into the required schema, and send it to the configured endpoint system. Here, it is important to again consider the semantics of "at-least-once" and "exactly-once" delivery and the requirements of the endpoint system. This means the Sink will need a buffer (rather, a temporary queue) to store Messages until they can be successfully delivered to the endpoint system, in the original order. The Sink may also need to filter duplicate data by using a message key, either by fetching one from its contents, or by using the Topic-Partition-Offset combination as a unique key.

6.1.4 Shared API

Although our microservices will be decoupled in nature, they will require a shared API for communicating. This API will be used by many—if not all—microservices in the platform architecture. For example, the Kafka Producer Client and the Kafka Consumer Client will require a set of constants defining the names of the Topics for each pipeline—which will allow

us to avoid manually adding/updating Topics each time a new integration is added; the shared API will also include helper functions for reporting to the Metrics server, and creating application logs in general; the API will facilitate communication with the App Logic component, and much more.

6.1.5 Client Area

The system will require a server where clients can create an account, access account settings, manage integration credentials, and create data flows. The Client Area will service clients from a Front-end, Node.js server (for simplicity's sake) containing the GUI in React (with minimal business logic), with the help of a Back-end REST API server in Node.js for handling requests. The Back-end server shall process requests from the Front-end (and any other interface built in the future) and route them to the App Logic component for committing changes in the user account contents.

6.1.6 App Logic Component

Our platform will need a server for handling various other parts of business logic, such as databases containing user information and integration credentials, and so forth. The App Logic component may be accessed by virtually any component of the platform, and for critical reasons, which is why this microservice must be designed in a scalable, distributed manner, with low latency and utmost availability. The related database will mostly store structured data, but must be designed with a distributed system in mind; we have opted for [#TODO add db here].

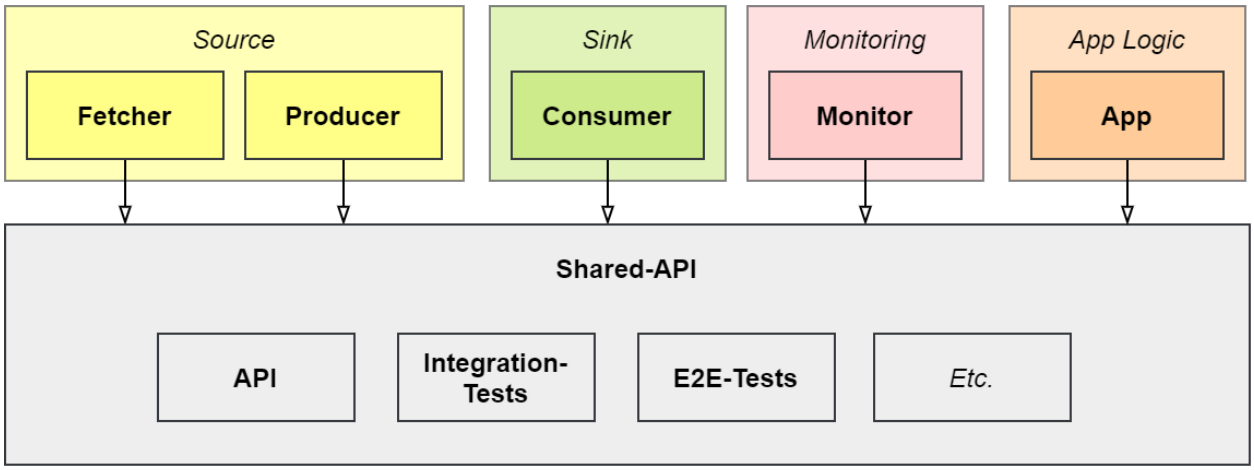

Illustration 6.2: Platform code organization

6.1.7 Testing

The maintainability of the platform, especially the development of future updates, will heavily depend on how well-designed the system was in the first place. In order to ensure the platform is working properly, there is an array of tests that can be conducted. **Integration tests,** as well as **end-to-end tests,** will ensure that our software produces the required results as a whole, in ideal conditions. We will use integration tests to validate design choices and fix flaws in the data flow which were not previously noticed. **Performance/"stress" tests,** using tools such as Netflix Chaos Monkey, will bring to light how well the platform works in real-life situations, and especially in difficult scenarios, such as high load, failure of the majority of nodes or complete "blackout" of clusters, downtime of microservices, bad network conditions, and so forth. Stress tests will help further improve the architecture and fault-tolerance of the system. **Unit tests** shall help create modular, independent microservices, with proper separation of concerns; unit testing will allow us to develop an architecture that is easy to manage as more components are built, and new integrations added down the line. This suite of tests, created in the early stages of the platform development, will ensure any maintenance team in the future can build modular code and manage the platform's production environment with ease, confidence, and a significantly lower risk of "breaking" the service.

6.1.8 Monitoring and Metrics

Building a data pipeline involves higher risk than some other types of web applications—there will be smaller consequences if a browser game goes down, than if a connector between a payment processor and a reporting application snaps. Therefore, our architecture must account for a metrics/monitoring server that will keep track of application health and its performance metrics. This monitoring service will be deployed as a standalone component, and will oversee critical metrics such as the number of events received by a webhook, the success rate of getting messages across the pipeline to Sinks, the availability of the Sink's temporary buffer, the frequency of Broker failures in the Kafka Cluster and how much time it takes to restart one, and so forth. Cloud systems typically make use of **internal and external** metrics: internal metrics are reported directly by the application itself (in our case, from the different components to the Metrics component), while external metrics, such as application availability or response time, are monitored via 3rd party systems (in our case, the Metrics component can be put in charge of gathering external measurements of the other components, or a 3rd party system can be employed for this purpose, giving the monitoring system more resilience in case the Metrics component itself goes down) [4]. These important statistics will be sent via our data pipeline from other components into the Metrics service as standard Messages, for further aggregation, processing, reporting and analysis. The Metrics server will offer an "at-a-glance" overview of the aggregated state of systems in order to provide a quick and non-overwhelming report of the system—in addition to more granular insights available for when deeper analysis is needed. In fact, the Metrics service will be one of the most important components of our platform in a commercial project, as it will give ongoing insights into the reliability and scalability of the architecture.

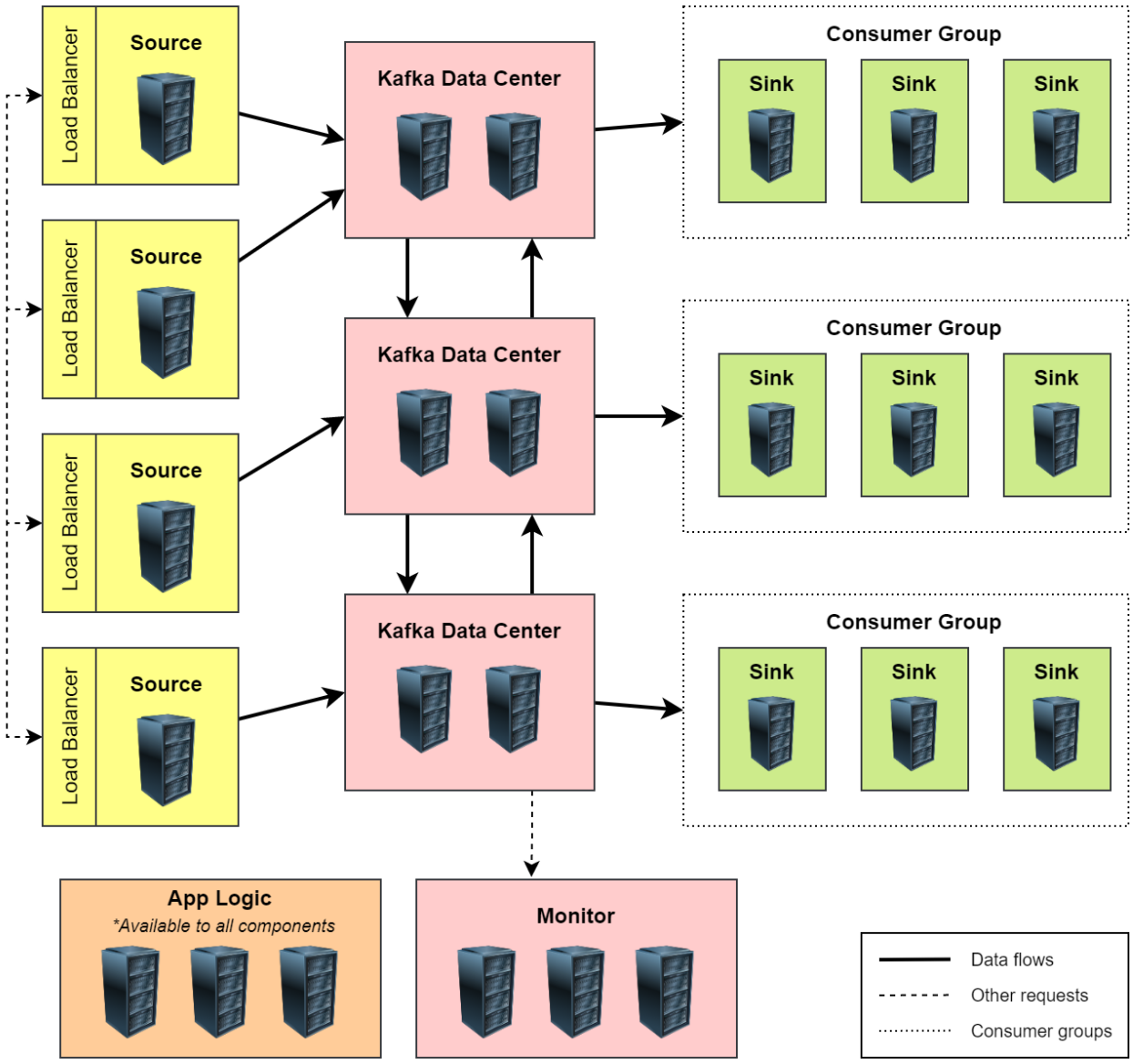

Illustration 6.3: Platform internal architecture. **Metrics data flows hidden.*

6.2 Documentation

6.2.1 Producer

The `Data-Pipeline-Platform.Producer` repository contains the code for the Producer component. The Producer is a Java Spring web server that accepts requests for further routing into the Kafka Cluster. This is an independent microservice that can be instantiated on any number of hosts, therefore providing high availability to servers sending data towards our platform, or other components making use of the Producer. The Producer is equipped with a load balancer to evenly distribute requests.

The `WebController` class represents a REST controller with a custom route for each source of data. For example, client's custom data may be sent via the `/api/web` route; Stripe notifications are accepted on `/api/stripe/`, and so forth. Each request is handled with a custom service from the `service` package that attempts to forward the data into the Kafka cluster, using the `KafkaProducer` class. Services are designed in line with Spring's programming paradigms; they are represented as interfaces and instantiated in the `impl` package. Most services represent subclasses of the `IProducer` abstract class. The `IProducer` class is a generic service with helper methods for instantiating the `KafkaProducer` through key properties such as `bootstrap.servers`, `key.serializer`, `value.serializer`, `acks` and more. Some properties of the producer instance are modifiable by the inheriting class (e.g. `IWebService`), while bootstrap servers and serializers are directly set by the `IProducer`, using externalized configuration variables [38]. `IProducer` implements the Kafka `Callback` interface—therefore all inheritors are required to at least support asynchronous message sending. This allows the Producer component to remain available for new requests without blocking system resources, while ensuring data delivery at the same time through the callback function.

6.2.2 Consumer

The `Data-Pipeline-Platform.Consumer` repository contains the code for the Consumer microservice. The Consumer is a simple Java server that listens to incoming messages using Kafka client's Consumer API, transforms the messages using Spark, and "forwards" the message into the receiving systems. Like the Producer component, this service too may be spawned on an arbitrary number of hosts with little configuration overhead, providing high availability to the Kafka Cluster.

The `Main` class receives arguments via the command line indicating which consumers to run (alternatively, no command-line argument means running all implementations). Each consumer is run on a separate thread. The `IConsumer` abstract class, and its subclasses, represent the building block of the Consumer package. The `IConsumer` class instantiates a `KafkaConsumer` object with the given properties; like its `IProducer` counterpart in the Producer package, `IConsumer` sets the most important properties by default—including the bootstrap servers configuration and serializers; the `IConsumer` also provides a default value for the offset reset config property. However, it is up to the subclass to set the group ID.

The `IConsumer.run()` method sets up the `KafkaConsumer` object by subscribing to the specified Topic and creating a simple while-loop that polls for the latest messages. Fetched records are left for processing to the implementing subclass via the `process` method. Every `IConsumer` class implementation can be subscribed to only one topic, by design; this means that every implementation handles one specific type of record in the `process` method. The `process` method is also where the Message is forwarded to the configured receiving systems of our platform, using Apache Spark.

For example, a user may have configured their account to receive Slack messages and send them to an email address. Slack will send their notifications to the Producer component, which funnels messages into a `SlackConsumer` sink; the Consumer will use Spark Streaming to process the received Message(s) and call a 3rd party service to dispatch the email.

7 Conclusion

7.1 Summary

Designing a platform to solve our problem statement involved several challenging steps. The first goal was to establish the exact issues our platform would be facing, such as authentication, data persistence, and so forth. The next step was to decide what level of reliability and availability would be promised; these characteristics had a profound effect on the architecture of the platform. Following this research,

7.2 Discussion

Cluster setup. It is debatable whether the setup of our Kafka Cluster is optimal. This proof of concept assigns one source system (Producer) to one Topic, which is then distributed to different endpoints (Consumers) upon transformation. However, it would be possible to create such a setup where a Topic represents a class of client accounts; for example, one Topic for free and lowest-tier accounts, one for mid-tier accounts, and another Topic for the most resource-heavy accounts. This would allow for more fine-grained monitoring and tuning of Brokers to give bigger accounts better SLA guarantees. Likewise, it would be possible to have entire Topics dedicated to individual accounts with the most extreme requirements.

Data accuracy. Kafka's architecture implicitly contains a possible risk of data loss or duplication, in case of either a Producer or Consumer failing. For example, if a Consumer fails and the commit offset is lower than the last message read, all messages between the offset and the last read message will be re-read, causing duplication. On the other hand, if the commit offset is larger than the last message read by the Consumer, the entire Consumer Group will miss the messages between the last read message and the offset. This design "flaw" is difficult to eliminate considering the distributed nature of our system. For this reason, our data pipeline platform is not an ideal solution for managing highly-sensitive data, such as purchase orders, as some events may get lost or duplicated. In general, Kafka does provide guarantees in terms of Message order—if Message A was received before message B, Consumers are guaranteed to read Message B after Message A [4]; and when it comes to data duplication, if the consuming application requires zero overlap, the timestamp of the message can be recorded into memory to ensure the Message was not previously processed. Another way to tackle the issue of data accuracy is by specifying custom retention and replication factors on a per-app basis; for instance, non-critical producers's designated Topics will allow for higher data throughput, lower reliability, and therefore higher risk of data loss; while producers with more sensitive data can be configure to use higher-redundancy Topics.

Configuration parameters. This article does not extensively test the exact configuration details of Kafka, Spark, and other technologies, but makes use of the publicly-available Docker images with the respective tools (mostly) pre-configured. However, the performance of this platform in a commercial project requires paying significant attention to the exact configuration parameters. A potential performance improvement opportunity lies in examining the impact of every source and destination application on the system, and distributing more cluster resources

towards performance-heavy systems, while saving resources on the lighter ones. This could be achieved with an elastic cluster, as well as by monitoring the load of sources/sinks and determining how to configure their respective data flow limitations. Furthermore, each of the Producer component's interfaces (e.g. for Stripe, GitHub, Slack) can be further optimized based on their event nature. Some producers should receive greater priority, while others may accept greater latency and even tolerate some loss of data. (In the case of MirrorMaker, running the process in Docker does not incur huge resource costs as it does not require a lot of memory; this is actually a popular way to instantiate MirrorMaker [4].)

Data serialization. By default, all services in the `Producer` component serialize data through the `StringSerializer` class. However, this may not be the optimal solution. Depending on the nature of the data, it may be possible to save bandwidth by creating custom serializers, although this will require more time and debugging. Apache Avro may also be introduced for easier schema management.

7.3 Future Work

Thanks to the use of modern design patterns, versatile architecture and flexible tools in the tech stack, our platform is well-positioned for future enhancements. Besides the possible issues mentioned in the previous section, here are ideas for improving the solution further by expanding its features:

Supported applications. The platform is currently able to integrate with a small number of 3rd party applications, which makes it a useful solution for a severely limited set of potential clients. A key goal for future development is to create interfaces for reading/writing to more platforms, thus expanding the product's functionality. The interfaces and classes related to integration, coupled with dedicated tests, should make it easy for the development team to add more platforms.

Data transformation features. It is possible to expand the platform's data transformation abilities, allowing users to combine multiple data sources into one stream. This expansion would give users the ability to define rules on how multiple data streams are to be merged before forwarding the merged events into the selected Sinks. For example, a data stream containing stock market data could be combined with a Twitter data stream in a certain time window, and forwarded to an application that analyzes the correlation of the events. Developing this feature may come with risks because multiple Topics are being read from, so extra care should be taken when committing the Messages as read.

Fetch frequency control. As some 3rd party systems require querying the client's data via a recurrent job, the platform has to determine when the query job should be performed. Some clients may prefer to manually tune when the data is fetched, which could prove a potentially highly useful feature, without adding too much complexity to the system. Clients may be offered the ability to set a recurring time interval, or even to create a timetable when their data is to be queried.

Automation capabilities. Workflow automation is a concept that has been growing in popularity since the invention of personal computers, and is based around defining rules that can be executed by a software to automate mundane or repetitive human tasks. "No-code automation" is especially gaining traction in the latest period; tools such as Zapier [39], Make

[40] and others are allowing users to create automated workflows without any coding. This is what makes the no-code automation concept similar to the Segment CDP platform [41]. In fact, our platform already supports a fraction of what a “no-code automation” tool could do. With the right modifications to the platform architecture, our simple Producer-to-Sink data flow could be transformed into a user-defined multi-step automation, using a similar tech stack. This would be a significant upgrade to the platform’s architecture, and will thus require lots of testing and caution during deployment.

8 References

[1] Wakefield Research. The State of Data Management Report.
https://get.fivetran.com/rs/353-UTB-444/images/2021-CDL-Wakefield-Research.pdf
(Dec. 2022)

[2] Segment.io, Inc. An introduction to Segment.
https://segment.com/docs/guides/
(Dec. 2022)

[3] Kulkarni, R. Big Data Goes Big.
https://www.forbes.com/sites/rkulkarni/2019/02/07/big-data-goes-big/
(Dec. 2022)

[4] Narkhede et al. *Kafka: The Definitive Guide.* O'Reilly Media, Inc, 2017.

[5] Wikipedia. Separation of concerns.
https://en.wikipedia.org/wiki/Separation_of_concerns (Dec. 2022)

[6] Wikipedia. Event-driven architecture.
https://en.wikipedia.org/wiki/Event-driven_architecture
(Dec. 2022)

[7] Red Hat, Inc. What is event-driven architecture?
https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture (Dec. 2022)

[8] Richardson, Chris. Pattern: Saga.
https://microservices.io/patterns/data/saga.html
(Dec. 2022)

[9] Hazelcast. What is Stream Processing?
https://hazelcast.com/glossary/stream-processing/
(Dec. 2022)

[10] Bifet et al. 2018. Machine Learning for Data Streams: with Practical Examples in MOA, The MIT Press

[11] Fernando, J. Moving Average (MA): Purpose, Uses, Formula, and Examples.
https://www.investopedia.com/terms/m/movingaverage.asp
(Dec. 2022)

[12] Apache Software Foundation. Apache Kafka.
https://kafka.apache.org/
(Dec. 2022)

[13] Apache Software Foundation. Powered by [Apache Kafka].
https://kafka.apache.org/powered-by
(Dec. 2022)

[14] Wikipedia. Apache Avro.
https://en.wikipedia.org/wiki/Apache_Avro
(Dec. 2022)

[15] Apache Software Foundation. Kafka protocol guide.
https://kafka.apache.org/protocol.html
(Dec. 2022)

[16] Netflix, inc. Chaos Monkey.
https://netflix.github.io/chaosmonkey/
(Dec. 2022)

[17] Apache Software Foundation. Welcome to Apache ZooKeeper™.
https://zookeeper.apache.org/
(Dec. 2022)

[18] Wikipedia. Apache ZooKeeper.
https://en.wikipedia.org/wiki/Apache_ZooKeeper
(Dec. 2022)

[19] Apache Software Foundation. ZooKeeper Programmer's Guide, Ephemeral Nodes.
https://zookeeper.apache.org/doc/r3.4.8/zookeeperProgrammers.html#Ephemeral+Nodes
(Dec. 2022)

[20] Apache Software Foundation. Unified engine for large-scale data analytics.
https://spark.apache.org/
(Dec. 2022)

[21] Amazon Web Services, Inc. Introduction to Apache Spark.
https://aws.amazon.com/big-data/what-is-spark/
(Dec. 2022)

[22] Databricks. Apache Spark™.
https://www.databricks.com/spark/about
(Dec. 2022)

[23] Vaidya, N. Apache Spark Architecture – Spark Cluster Architecture Explained.
https://www.edureka.co/blog/spark-architecture/
(Dec. 2022)

[24] Apache Software Foundation. Cluster Mode Overview.
https://spark.apache.org/docs/latest/cluster-overview.html
(Dec. 2022)

[25] Wikipedia. Apache Spark.
https://en.wikipedia.org/wiki/Apache_Spark
(Dec. 2022)

[26] Docker, Inc. Docker overview.
https://docs.docker.com/get-started/overview/
(Dec. 2022)

[27] Flexera, Inc. State of the Cloud Report.
https://resources.flexera.com/web/pdf/Flexera-State-of-the-Cloud-Report-2022.pdf
(Dec. 2022)

[28] Docker Inc. Build and Ship any Application Anywhere.
https://hub.docker.com/
(Dec. 2022)

[29] Berggren et al. 2022. Differences in performance between containerization & virtualization with a focus on HTTP requests

[30] Docker Inc. Docker Engine overview.
https://docs.docker.com/engine/
(Dec. 2022)

[31] Baeldung. Virtualization vs Containerization.
https://www.baeldung.com/cs/virtualization-vs-containerization
(Dec. 2022)

[32] Wikipedia. System image.
https://en.wikipedia.org/wiki/System_image
(Dec. 2022)

[33] Docker Inc. Ubuntu. https://hub.docker.com/_/ubuntu
(Dec. 2022)

[34] Docker Inc. Packaging your software.
https://docs.docker.com/build/building/packaging/
(Dec. 2022)

[35] Docker Inc. Dockerfile reference.
https://docs.docker.com/engine/reference/builder/
(Dec. 2022)

[36] Docker Inc. Docker pull.
https://docs.docker.com/engine/reference/commandline/pull/
(Dec. 2022)

[37] Docker Inc. Continuous integration with Docker.
https://docs.docker.com/build/ci/
(Dec. 2022)

[38] Webb et al. Spring Boot reference guide, 24. Externalized Configuration.
https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/boot-features-external-config.html
(Dec. 2022)

[39] Zapier Inc. Zapier.
https://zapier.com/
(Dec. 2022)

[40] Celonis, Inc. Make.
https://www.make.com/en
(Dec. 2022)

[41] Segment.io, Inc. Event-driven architecture, Kafka and CDPs: Joining internal infrastructure with your tech stack. https://segment.com/blog/kafka-and-cdps/ (Dec. 2022)