Restructuring microservice systems with ordered messaging and parallel process flow approaches for performance optimization

Julia Sulina

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 26.07.2022

Supervisor

PhD Lachlan Gunn

Advisor

MSc Jean-Luc Lamadon

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Julia Sulina

Title Restructuring microservice systems with ordered messaging and parallel process flow approaches for performance optimization

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor PhD Lachlan Gunn

Advisor MSc Jean-Luc Lamadon

Date 26.07.2022 **Number of pages** 60 **Language** English

Abstract

Microservice systems are composed of small services with a single responsibility, which can be independently developed and scaled. The design has the disadvantage of increased system complexity because of the introduced remote communication for service interaction. Communication patterns strongly influence microservices system architecture and performance. They strengthen or weaken the independence of services, contributing to system availability and reliability. The system's load increases with the introduction of new users, but the system's performance still needs to meet agreed service-level objectives.

The thesis aims to analyze the impact of changes in communication patterns and technologies on the microservice system performance on the example of the Connected Retail Inbound pipeline. Performance is defined in terms of latency and throughput of the system fragment. The study describes two approaches for enhancing the performance of microservice systems by restructuring: ordered messaging and parallel process flow. The work is limited to trialing the approaches on only one microservice system. Theoretical sources suggest that both techniques would improve the performance. However, the implementation of the ordered messaging method had an adverse effect on the pipeline message processing performance in terms of throughput. The parallel process flow approach improved the pipeline performance in terms of the average worst message processing latency. Queueing theory models are used to evaluate observed results.

Keywords Microservices, communication, queueing theory, performance

Contents

Abbreviations

Acronyms

API Application Programming Interface. 7, 12, 13, 19, 20, 26–29, 31, 33, 34, 40, 43, 45, 47, 50

AWS Amazon Web Services. 20, 21, 26, 27, 30, 31, 35, 40, 41, 49, 51

CPU Central Processing Unit. 28

CR Connected Retail. 26–28, 30, 31, 33

CSV Comma-Separated Values. 31

FIFO First-In-First-Out. 21, 31, 35

HTTP Hypertext Transfer Protocol. 19

REST Representational state transfer. 13, 19, 26–28, 33, 40, 43, 45, 47

S3 Simple Storage Service. 19, 26, 31, 32, 35, 49

SLO Service-level objective. 26, 43, 44

SNS Simple Notification Service. 40, 41, 45

SOA Service-Oriented Architecture. 10

SQS Simple Queue Service. 20, 21, 27, 28, 30, 31, 33, 35, 37–39, 41, 42, 44, 45, 47–49, 51

TCP Transmission Control Protocol. 19

Acknowledgements

I would like to thank my supervisor, Lachlan Gunn. I would also like to express my gratitude to my advisor, Jean-Luc Lamadon, and the team *Fuel* at Zalando Connected Retail, who supported me throughout this project. Finally, I would like to thank my family, who endured this long process with me.

1 Introduction

Microservice software systems enable developers to deploy and change system components independently, which makes development easier. However, services need to communicate to perform larger operations. Messaging patterns define the extent of service coupling. Availability of services using synchronous communication often depends on the availability of the service they are calling. Asynchronous communication does not fit all the possible messaging requirements and usually requires building a reliable cache on the calling system. This work aims to analyze the impact of changes in communication patterns and technologies on the microservice system performance in the Connected Retail Inbound pipeline example.

The system's load increases with the introduction of new users, but the system's performance still needs to meet agreed service-level objectives. The pipeline in its original state had several bottlenecks. A considerable number of write operations to databases in many services are caused by unordered messaging, a legacy service performing expensive database queries, and synchronous service calls.

Two communication-related methods to restructure the existing microservice system are assessed to improve pipeline performance. Performance is defined in terms of latency and throughput of the system fragment. The first approach orders article messages to enable microservices optimizations and improve performance. The second approach offloads less time-critical operations from the main flow introducing parallel process flow and removing synchronous API calls. The techniques are evaluated using the queuing theory model and concrete implementation experiments.

2 Background

2.1 Microservice architectures

With a microservice architecture, software applications are split into smaller, simpler functioning components that communicate with each other via network requests (Zeng et al., 2022). Although microservice architecture helps reduce the unit complexity by enabling high cohesion and low coupling, it requires communication platforms due to its distributed nature (Karabey Aksakalli et al., 2021).

Scalability and independence are critical features of microservice architectures that facilitate system reliability of the services (Hasselbring and Steinacker, 2017). Scalability is the ability of the system to tolerate increasing load by utilizing additional resources (Herbst et al., 2013). Decoupling means reducing dependencies between system components. Dependency between the services in a microservices system shows a possibility of independent altering of the component and separate updates and deployment of the system element (Kul and Sayar, 2021). Independence is the ability of the service to perform its function regardless of the availability of other components. Yuan (2019) sees the advantage of microservices architecture in the suitable separation of concerns. Differently from monolithic systems architecture of microservices obligate to consider two levels ‘micro‘ as a design of the service and ‘macro‘ as a collective result of the interaction of the services (Yuan, 2019). Each microservice is built to fulfill specific business capabilities and should support independent, fully automated deployment (James Lewis and Martin Fowler, 2014). To conclude, strong independence in terms of functioning and deployment (easing maintenance), system scaling, and ability to recover are the most widely agreed beneficial aspects of microservices in cloud environment (Taibi et al., 2018). Centralized management of the microservices is minimal (James Lewis and Martin Fowler, 2014). The services might be written using different programming languages and utilize various technologies for data storage (James Lewis and Martin Fowler, 2014). Abstractly, a microservices system can be represented as multiple components and connectors, with distinct sets of component types and connector types, where indirect and direct dependencies can be described as a different set of connectors (Ntentos et al., 2021). Such an abstraction makes it simpler to analyse and compare the service system complexity and dependencies.

Microservices cloud deployments using cloud resource allocation on demand allow companies to reduce infrastructure costs compared with monolithic applications. Large applications are developed as a set of smaller components, allowing independent

implementation by separation into independent code bases managed by small teams. Components can be operated and deployed independently to propagate incremental changes. The benefits of agility, cost reduction, and granular scalability of microservice architecture come with development efforts, technical challenges, and costs of adopting new practices, processes, and methodologies. Microservice architectures are commonly compared to monolithic services. Monolithic applications are defined as larger multi-functional services. According to Villamizar et al. (2017) case study, the monolithic approach is faster and more practical for applications with a small number of users (up to thousands of users). Later monolithic applications could be incrementally modified to microservices to improve the scaling of services and teams. (Villamizar et al., 2017)

Microservice architecture, as distributed system, have several problems: operational complexity, communication, dependence between services, availability, and scalability (Kookarinrat and Temtanapat, 2016). With an increase in the number of nodes in the system, operational complexity increases; as a result, system maintenance tools for monitoring, service discovery, and configuration management become inevitable. Communication between separate services requires remote connections; to maintain performance with high network communication latency, microservices need lightweight communication infrastructure (Kookarinrat and Temtanapat, 2016). Communication might increase dependencies between services; for example, point-to-point communication or static connections make services hard to evolve and decrease the flexibility of the system (Kookarinrat and Temtanapat, 2016). Connections like message bus decrease coupling but, on the other hand, might become the single point of failure for the system (Kookarinrat and Temtanapat, 2016). To increase independence, usage of message brokers-based communication has been suggested by Christudas (2019). If a consumer does not have issues and is running fast enough, it will receive messages almost immediately, making the performance of such communication comparable to the synchronous communication model (Christudas, 2019). On the other hand, if a consumer is not able or too slow to process the messages, the message broker stores them until being processed by all subscribed microservices (Christudas, 2019).

Designing a microservice-based system is a much more challenging task than designing a single service. There is a lack of well-defined methods proposed for the analysis and design of microservice-based architectures (Unlu et al., 2021). However, widely adopted architecture patterns exist that, to some extent, fulfill at least some microservice characteristics. Valdivia et al. (2020) concludes that the most common

microservice architecture patterns aim to target issues in communication or optimize microservices system performance (Valdivia et al., 2020). Among the valued features, the most frequent are maintainability and reliability (Valdivia et al., 2020).

One of the methods effective for strengthening services autonomy is using event-driven architecture (Indrasiri and Siriwardena, 2018). An event-driven architecture is a software architecture pattern based on the ability of applications to detect and respond to events that correspond to a significant change in the state of a system or its environment (Chandy, 2009). Frequent patterns for an event-driven architecture are event notification, event state transfer and event sourcing (Indrasiri and Siriwardena, 2018). Event-driven communication patterns are generally implemented with the message queuing and publisher-subscriber messaging infrastructures (Indrasiri and Siriwardena, 2018). An event-driven communication pattern is an example of a pattern focused on communication. Besides communication, different microservices architecture patterns focus on other aspects of the services including behaviour, deployment, and data storage (Taibi et al., 2018).

One of the examples of an behavioural pattern is the *chain of responsibility* design pattern. In this pattern, a collection of sub-services are designed to process requests sequentially; output from one service becomes an input for the other one (Akbulut and Perros, 2019). Deployment and data storage patterns usually differ according to the number of services or data storage components per unit: for deployment, it might be one or multiple services per host, and for data storage, one database per service, cluster, or whole system (Taibi et al., 2018).

Microservice architectures are an extension of Service-Oriented Architecture (SOA), which similarly to microservices, uses messaging communication and components modularization. However, there are essential differences in communication, data, and service access. Microservices communicate with lightweight network requests instead of a central event bus, data is decentralized, and access to service functions is independent. (Zeng et al., 2022)

The primary way of componentizing the microservices software is by breaking it down into services that can be independently deployed. This way, services as components have explicit interfaces, which are lacking for libraries in most languages. (Bakshi, 2017)

Design principles for microservices like single responsibility of service, small service size, absence of a shared database, and independent release process (Megargel et al., 2021) closely relate to the commonly named microservice advantages. Small and independent services are seen as an advantage as the internal implementation of the

service does not influence other system components, development and deployment of new features do not require changes in the entire system (Richter et al., 2017). Interruptions in one service operation do not interrupt the entire system (Unlu et al., 2021). Loose coupling between microservices allows testing and deploying independently; when interfaces are preserved, services can be replaced by other versions easing replaceability and versioning (Richter et al., 2017) (also referred to as evolvability (Bogner et al., 2021)). On the other hand, applications require the ability to tolerate the failure of services, as unavailability of the infrastructure may cause service call failures (Bakshi, 2017). Microservices using horizontal scaling and a load balancer distributing the requests to available service instances must be stateless and not share a database with other microservices (Richter et al., 2017).

Disadvantages of the microservices approach mainly relate to complexity. The system's complexity is no longer encapsulated in the service implementation but is spread to the execution environment. Such complexity leads to challenges with monitoring and testing the system behavior. Communication overhead and data consistency are additional challenges for microservices: remote communication introduces latency overhead, and eventual consistency is a common strategy in microservice architectures. (Richter et al., 2017).

2.2 Communication

Due to the distributed nature of microservices, parts of the system need to employ some communication strategy to stay connected (Karabey Aksakalli et al., 2021) communicating with each other via network requests (Zeng et al., 2022). A system or a product is composed of interactions between the microservices (Indrasiri and Siriwardena, 2018). Therefore, inter-service communication is crucial for microservices architecture success (Indrasiri and Siriwardena, 2018). When designing the communication architecture of microservices, it is necessary to mitigate the dependencies of services on each other as much as possible (Singh and Peddoju, 2017). When the service dependencies cannot be eliminated, the advantage of using microservices is lost (Karabey Aksakalli et al., 2021). In this context, approaches are needed to reduce dependencies when designing microservices' communication architecture (Karabey Aksakalli et al., 2021).

Message format to exchange information and service collaboration mechanism should be chosen appropriately for microservices architecture, based on the use case (Indrasiri and Siriwardena, 2018). Two central design patterns for service collaboration are choreography and orchestration (Megargel et al., 2021). Choreography

pattern implies asynchronous communication without a central controller, where events are published using a message broker and microservices are responsible for subscribing to the events and processing them (Megargel et al., 2021). In orchestration pattern controller - composite microservice solely responsible for the application flow synchronously invokes other microservices carrying out steps needed for completing the application process (Megargel et al., 2021). When the orchestration model provides better flow visibility than choreography, choreography solutions result in less coupling and less communications (Megargel et al., 2021). Solutions may also use orchestration and choreography combined approach (Megargel et al., 2021).

Request-response communication patterns can be divided into synchronous and asynchronous communication. During synchronous communication, the client waits for a response from the service to proceed (Karabey Aksakalli et al., 2021). The call fails if a response is not received before the predefined timeout duration (Karabey Aksakalli et al., 2021). Application Programming Interface (API)-based communication follows a synchronous and request-response messaging type of communication (Indrasiri and Siriwardena, 2018). Synchronous requests to remote services are likely to make the system slower; moreover, such requests might cause cascading system failures if the service depends on the data or functionality of the called service (Janssen, 2017; Karabey Aksakalli et al., 2021). During asynchronous communication, the client is not blocked waiting for a response; it uses asynchronous media such as a messaging queue to send a request and later receive a response (Karabey Aksakalli et al., 2021). The asynchronous messaging design pattern is preferable for processing large volumes of data without needing an immediate response (Akbulut and Perros, 2019). The asynchronous messaging design pattern is most used as a communication model between microservices; it can be combined with any other pattern (Pacheco, 2018). Asynchronous communication not often follows the request-response paradigm. For example, Pacheco (2018) describes a *purist microservice* as a microservice that receives requests and performs its task without the need to return a response; besides, it should function without requiring communication with other microservices. The asynchronous messaging design pattern naturally supports such communication (Pacheco, 2018). Even if the response is delivered, it might not mean that the request processing was completed rather than just started (Pacheco, 2018). Such an approach is helpful from a scalability point of view and can be used by applications with a high level of resilience (Pacheco, 2018). Furthermore, the pattern is convenient for the asynchronous character of microservices as it allows to perform the tasks without making application consumers wait (Pacheco, 2018).

Some authors suggest a framework capable of switching between synchronous and asynchronous communication models based on system load (Sriraman and Wenisch, 2018; Shadija et al., 2017). In practice, this is not usually done, and communication between services is fixed on runtime.

Point-to-point communication is a direct communication model where each service can call other services, for example, using Representational state transfer (REST)ful API. However, the drawbacks of point-to-point-based communication are performance degradation and the increased complexity of services for larger systems. Besides, the increased number of development, operation, and maintenance point-to-point communication models get exponentially more complex. (Karabey Aksakalli et al., 2021)

Publish-subscribe communication pattern enables instant event notifications for microservice-based applications and supports an event-driven architecture (Karabey Aksakalli et al., 2021). Such communication is asynchronous; it does not expect any response, the events are broadcast to subscribers of the topic responsible for consumption (Karabey Aksakalli et al., 2021). Reasons to use message broker-based communication are the loose coupling of services, independent processing and buffering of messages, and scaling of communication (Whitesell et al., 2022). Loose coupling means that the sender of a message does not have to be informed about any characteristics of the microservices receiving the message (for example, endpoints of the receiver) (Whitesell et al., 2022). Independent processing of messages allows microservice architecture solutions to change over time (Whitesell et al., 2022). Buffering messages helps to avoid adverse effects from service interruptions, as message brokers utilize queues to buffer the messages delivered when consumers can process them again (Whitesell et al., 2022). Finally, the communication method is scales, achieving higher throughput and availability than other methods (Whitesell et al., 2022). Comparing message-broker (RabbitMQ) based communication with RESTful API method Hong et al. (2018) found that RESTful API response speed is better in case of a small number of users. However, the message queue performed more stably when the number of users increased, whereas the RESTful method had some error responses (Hong et al., 2018).

The persistence of messaging for interservice communication significantly affects the system's quality (Ntentos et al., 2021). Therefore, it is perceived as a violation of microservice best practices if services communicate without using an intermediary component that can persist the communication and no persistent messaging occurs between them (Ntentos et al., 2021). Some examples of communication persisting are

the usage of message brokers, a persistent publish-subscribe model, event sourcing, or database (Ntentos et al., 2021). Event sourcing is based on the persisting state-changing events in sequential order in the event log or event bus (Indrasiri and Siriwardena, 2018). Subscribers can restore the entity's state by processing the sequence of events for the entity (Indrasiri and Siriwardena, 2018).

Due to microservices' scalability, parallelism of the message processing multiply increases, improving the performance of the service, but at the same time results in the high concurrency of processing and out-of-order events (Oliveira Rocha, 2022). Implementation cannot assume that the events are ordered, that is why commonly event versioning is used to check order of event on the arrival (Oliveira Rocha, 2022). Out-of-order events can occur in multiple parallel processing cases for example on the message retries, when processing failed first time. Such events can jeopardize the consistency of the system (Oliveira Rocha, 2022). According to Oliveira Rocha (2022) the best approach to deal with concurrency is to partition requests end-to-end so that the parallel processing of the same type of events becomes impossible. With end-to-end partitioning greater performance can be achieved while completely avoiding concurrency and ordering problems (Oliveira Rocha, 2022). However, the data distribution between the partitions can become largely unbalanced if there is a large number of events routed to the same partition (hotspotting) (Oliveira Rocha, 2022). Although communication technologies can provide ordered messages, utilising them needs to cover end-to-end ordering which might provide design challenges in order to avoid concurrency.

2.2.1 Publish–subscribe

Loosely coupled and distributed microservices architecture has influenced the growth of demand for asynchronous communication and the popularity of the publish-subscribe approach (Kul and Sayar, 2021). Publish–subscribe pattern is a communications method where messages produced by publishers (senders) are grouped into topics. In this case, messages are not addressed directly to individual recipients; publishers are unaware of subscribers to their messages. On the other hand, subscribers might choose any topic and only receive communications meaningful to them without any knowledge of the publishers. (Kul and Sayar, 2021; Kul et al., 2021)

Decoupling using a publish-subscribe communication pattern supports changes in any aspect of publisher and subscriber services as far as the publisher can produce and subscribers consume messages in the same format as previously. In such a case, neither publisher nor subscriber has to know of any development on the counterpart. (Kul

and Sayar, 2021)

Event-based communication relies on the eventual consistency principle. A microservice broadcasts an event upon a significant change, such as a modification in a business entity. Other microservices subscribed to such events when receiving it might perform their logic like changing their entities or publishing a new event. Such a publish-subscribe pattern is usually powered by event bus technology. (Kul and Sayar, 2021)

In a client-server architecture, the client calls the service's endpoint directly. The publish-subscribe model introduces an intermediate component—the message broker—which links publisher and subscriber. The message broker transmits and filters messages, delivering them to correct subscribers. (Kul and Sayar, 2021)

Publish-subscribe model functionality is built around a topic—the publisher sends messages to the topic, which publishes messages to all subscribers immediately. Unlike typical message brokers, which expect users or services to request and retrieve the messages, with the publish-subscribe method, event-driven services are supported without polling a message queue for messages. Multiple services can consume the same message this way in parallel. (Kul and Sayar, 2021)

2.2.2 Kafka

Apache Kafka is an event streaming platform that implements the publish-subscribe communication pattern (Kafka documentation, Chapter 1.1). Kafka events can be processed immediately or later, as storing events for a defined retention period is supported (Milosavljević et al., 2021). Apache Kafka is an elastic, distributed, highly scalable, fault-tolerant platform (Milosavljević et al., 2021). The Kafka platform enables messaging, processing, and storage of events, which allows solving multiple common technical challenges using one technology cost-efficient way (Liermann and Stegmann, 2021, p. 74).

Kafka provides the real-time stream of messages in topics (Kul et al., 2021). Messages are produced and sent to topics by publishers and read from topics by subscribers (Kul et al., 2021). Kafka follows a common design for messaging systems, where data is pushed to the broker by the producer and pulled from the broker by the consumer (Kafka documentation, Chapter 4.5). A topic may be divided into one or several partitions (Wu et al., 2019). Partition is the physical storage of messages in the Kafka cluster, each partition is replicated across multiple nodes for fault tolerance (Wu et al., 2019). Partitions are used in Kafka for scaling a topic across many servers to facilitate parallel reading and writing the messages (Wu et al., 2019).

Multiple producers can write to the same topic or partition at the same time (Fahland, 2019). If multiple producers publish messages to the same partition, the messages will be written in the order of arrival (Fahland, 2019). The partition for the message must be specified by the publisher when it is written (Fahland, 2019). The producer can publish a message to a randomly selected partition or a partition determined by a partitioning key and a partitioning function (Auradkar et al., 2012). If the partition is assigned using a partitioning key, the partition number for a message is determined by hashing the key divided by the total partitions count. Users can also provide their own partitioning function for message distribution to partitions. Messages are partitioned by the selected key hash by default; this way, all the messages with the same key are saved to the same partition (Fahland, 2019). Different from hash partitioning, random partitioning messages are spread evenly between partitions using a round-robin partition assignment which results in a similar load for consumers and makes scaling the consumers easier (Boyle, 2021; Dhanushka, 2021). For a small set of keys hash partitioning results in unequal work shares (Korab, 2015). With a small number of partitions, the load might be so different that some consumers might get more messages and some consumers might experience starvation (Korab, 2015). Partitioning can be balanced with providing own business-specific partitioning function (Korab, 2015).

A consumer group consists of one or more consumers that receive messages from subscribed topics together, meaning that a message is delivered to only one of the consumers from the group (Auradkar et al., 2012). This way consumers from different consumer groups can process the same messages multiple times without the need to duplicate data (Milosavljević et al., 2021). Consumers of messages can form a consumer group; in this case, each consumer reads from a subset of partitions in the topic (Fahland, 2019). The smallest unit of parallelism for consumption in Kafka is a partition of a topic (Auradkar et al., 2012). Such parallelism implies that at any given moment, messages from one partition are only consumed by a single consumer within each consumer group (Auradkar et al., 2012; Fahland, 2019). Consumers can read messages from multiple partitions of the topic, but one partition might be consumed by only one instance within the consumer group (Wu et al., 2019) as illustrated in Figure 1. Consumers belonging to the same consumer group may be from different processes or different machines (Auradkar et al., 2012). The aim is to avoid consumers' coordination overhead while dividing messages evenly (Auradkar et al., 2012). The number of partitions controls the maximum parallelism within the consumer group; if the number of instances in the consumer group is larger than the

number of partitions in the subscribed topic, there will be idle consumer instances (Wu et al., 2019). For better load balancing, an approach known as *overpartitioning a topic* can be used; in this case, many more partitions in a topic than the consumers in each group are created (Auradkar et al., 2012).

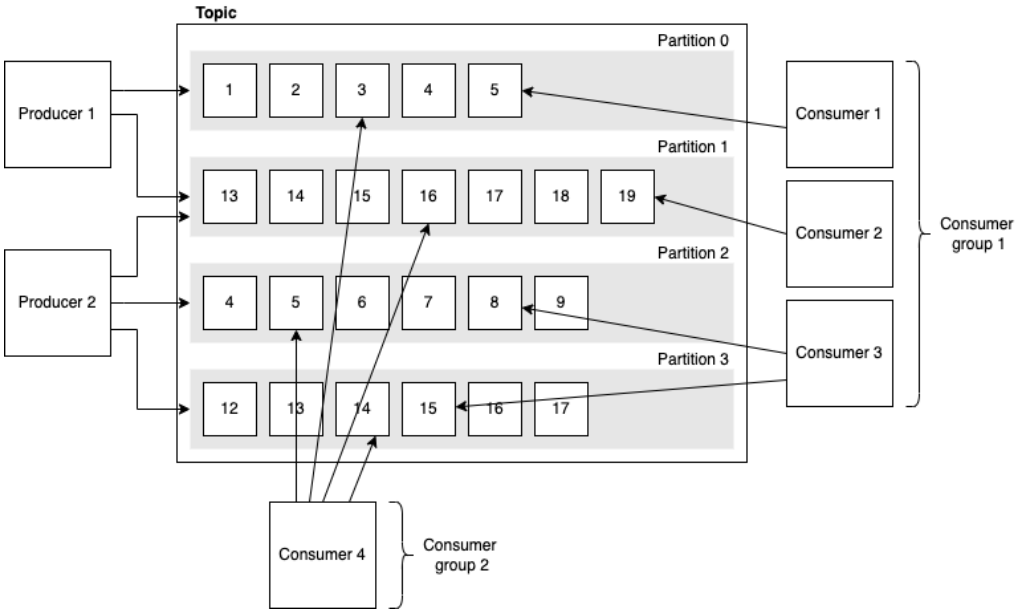

Figure 1: Kafka topic divided into partitions. Producers publish messages to partitions. Consumers keep track of consumed messages and form consumer groups.

Unlike many other messaging systems, a consumer should keep track of the messages consumed in Kafka. As the broker does not track information about message consumption for each subscriber, it reduces the complexity of the broker and the overhead of maintaining such data (Auradkar et al., 2012). However, this means that a broker is unaware if a message was consumed by all subscribers and can be deleted (Auradkar et al., 2012). Consumers also do not delete the messages. Instead, Kafka has a retention time policy, a mechanism deleting messages automatically after a configured period of time (Auradkar et al., 2012; Fahland, 2019). If necessary, messages can be also stored as long as needed, which provides durability (Bayramcavus et al., 2021); long-time retention does not affect the performance of Kafka (Milosavljević et al., 2021). Consumers can roll back to some previous offset and re-consume the data, which is a helpful feature for many consumers not common for traditional queues (Auradkar et al., 2012).

A message within a topic is uniquely identified by the partition and offset—the message sequence number (Fahland, 2019). A partition is therefore an immutable sequence of messages which can not be divided across brokers (Wu et al., 2019).

Kafka guarantees strict message ordering within a single partition, meaning that all consumers reading a partition receive all messages in the same order as they were written to the partition (Fahland, 2019). There is no ordering guarantee between messages in different partitions or different topics (Fahland, 2019). The number of partitions can hardly be changed at runtime (Henning and Hasselbring, 2021).

Periodically, consumers commit to the current processing position of a data stream, which means saving an offset of the message last processed. The frequency of committing significantly impacts the event-time latency of stream processing application (Henning and Hasselbring, 2021). Consumers may apply two strategies for committing offsets: commit-first and process-first (Fahland, 2019). The commit-first strategy provides an at-most-once processing guarantee, as the message would not be delivered again even if processing failed (Fahland, 2019). The process-first strategy provides at-least-once processing guarantee, as the offset is not acknowledged if an error in processing is encountered (Fahland, 2019). With a process-first strategy, the consumer may get blocked trying to reprocess same message again and again if they encounter faulty messages. Also, rereading messages may result in duplicates (Fahland, 2019). On the other hand, with a commit-first strategy, some messages might be unnoticeably skipped. In Kafka, a consumer reads messages from a partition only sequentially (Auradkar et al., 2012). If a particular message offset is committed, it indicates that the consumer has processed all messages before that offset in the partition (Auradkar et al., 2012).

Separate consumer groups are independent; each consumer can read messages at its own speed (Fahland, 2019). Such independence is essential for a distributed system because it supports decoupling and increases tolerance against slower components (Fahland, 2019). Kafka is used in large social media platforms like LinkedIn (Auradkar et al., 2012), but also for machine learning infrastructure and may benefit areas including inference of the analytic model in real-time, monitoring, and alerting (Kul et al., 2021).

Increasing the Kafka number of partitions can significantly improve its throughput and improve performance (Dobbelaere and Esmaili, 2017; Wu et al., 2019), because message consumers count is bound to the number of partitions. Kafka is provided as a service by cloud vendors. Kafka configuration parameters such as batch size, partition number, and log retention time contribute to achieving the best performance for a specific application (Wu et al., 2019).

2.2.3 Nakadi

Nakadi is a Kafka-based event-bus messaging system developed at Zalando and open-sourced. Nakadi aims to provide an event broker infrastructure with low latency event delivery via a secured RESTful API, thus increasing convenience for event-driven applications and asynchronous microservices development (Nakadi Manual). Nakadi acts as an HTTP proxy for Kafka, adding some functionality on top of Kafka and enabling the use of Kafka streams via RESTful API. When Kafka uses a binary protocol over Transmission Control Protocol (TCP) (Kafka protocol). Both producer and consumer can communicate with Nakadi using Hypertext Transfer Protocol (HTTP) requests. Nakadi uses the concept of event type, an abstraction of Kafka's topic. Nakadi functionality includes event type schema registry, validation of event types, and access control for event types. Validation of events against a published schema ensures that consumers can process the event. When event type is created, it is configured by setting read and write partitions. To consume events, consumers should create a subscription for the event type. Nakadi keeps track of consumed events offset of consumer group within the subscription. Nakadi inter-service messaging is traceable, as every event is archived. (Nakadi Manual)

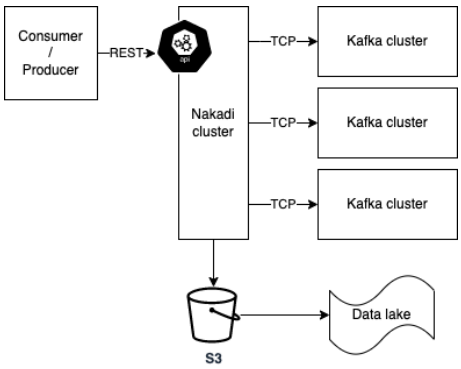

Figure 2: Nakadi proxy exposing RESTful API and archiving events in S3 bucket.

Nakadi preserves the arrival order of events; however, events sent from the producer may arrive differently from the original order due to network latency. If there is a need for end-to-end event ordering, that case producer needs to ensure this itself. For example, the producer may wait for a response from Nakadi before publishing the next event or define ordering semantics in event type so that the consumer can restore the order of the events. Events are then only ordered within the partition. Services that require ordering for the same entity updates need to configure a hash partitioning strategy for their event type and select a field to construct the key. Total ordering of the events is impossible with Nakadi (or Kafka) unless the

topic has only one partition. Total ordering is not commonly required or desirable as it can limit system scalability and result in hot spotting. (Nakadi Manual, Event Ordering)

The number of partitions in an event type for Nakadi is configurable on the creation of the event type and cannot be modified. Event throughput scaling depends on a number of partitions. Increasing throughput by creating a new event type might be difficult as it requires coordination between producers and consumers to take the new event into use. It is recommended to create more partitions than actually needed so that each consumer initially reads from multiple partitions; for even workload distribution partitions number for each consumer should be the same. Then the number of consumers can be increased as throughput increases until it reaches the number of partitions. (Nakadi Manual, Recipes)

Nakadi relies on many features of Kafka; for example, it supports log compaction. Log compaction is a mechanism that ensures that in a single topic partition, at least one last state for each key will remain (Kafka documentation, Chapter 4.8). There are some differences, too; Nakadi exposes event types instead of topics in its API. Event types define event structure and stream configurations. Besides, consumers receive messages in batches instead of individual messages (Nakadi Manual). API abstraction makes it easier to manage upgrades to systems like Kafka as it is not a direct dependency of microservices (Nakadi Manual). Nakadi is a complement to Kafka; it allows the use of Kafka without introducing a global dependency (Nakadi Manual). Asynchronous event delivery is a more straightforward option for a microservice architecture than synchronized calls requiring mitigation with caches and circuit breakers (Nakadi Manual).

2.2.4 SQS

Nakadi and Kafka are different from queues such as Amazon Web Services (AWS) Simple Queue Service (SQS). The message in an Simple Queue Service (SQS) queue can only be consumed once. For events to be consumed by multiple consumers, every consumer commonly uses a dedicated SQS queue. An SQS queue does not store already consumed messages, so messages cannot be "replayed" again like from a Kafka stream. Messages must be explicitly removed from the queue by a separate call when consumed. The message is automatically retried if it is not removed from the queue within a configured timeframe. After a configurable number of unsuccessful deliveries, the message is moved to a dead letter queue. (AWS SQS)

The SQS approach of confirming individual messages has advantages compared

to the Nakadi or Kafka approach of a consumer advancing a cursor in the data stream (AWS SQS). If a single message is unprocessable, for instance, because of an invalid format, in SQS, only this message is delayed. With Nakadi or Kafka cursor semantics, the client has two options: stop further message processing until the problem is resolved or skip the message and move the cursor (AWS SQS). SQS queues by default do not guarantee ordering, however the AWS SQS queue type First-In-First-Out (FIFO) provides ordering per messages group id (Gregory, 2021).

2.3 Microservice performance optimization

A study by Singh and Peddoju (2017) shows that microservice-based application performance in terms of response time and the throughput can be better than that of a monolithic application (Singh and Peddoju, 2017). The simplest method to assess the success of microservices is to compare their performance with the performance of the monolithic implementation; the performance of the microservice-based system should meet or exceed the pre-migration performance (Akbulut and Perros, 2019).

There are many ways to improve microservice system performance: optimizing performance of the individual microservices service times, minimizing service dependencies, enhancing communication links and reducing queueing time, adding redundant service instances, and load balancing bottleneck services. To increase individual microservices' performance, standard service enhancements like improvements in the code or optimizing database queries apply. An example of minimizing service dependencies might be removing the dependence on a database or minimizing API calls to other microservices. The number and nature of requests in a microservice system affect performance, so to achieve performance gains, the communication model should be assessed for potential improvements (Karabey Aksakalli et al., 2021).

Another tool for performance optimization is load sharing, conceptually similar to load balancing, which became accessible on the service level and easy to implement using cloud infrastructure. Eager et al. (1986) discussed the load-sharing concept in terms of locally distributed systems related to computational resources sharing. In the system where work generated at one node stays there for the whole processing, there is a possibility of uneven load distribution, which lead to poor overall system performance (Eager et al., 1986). Redistribution of the load among the nodes, load sharing, aims to improve performance (Eager et al., 1986). Load balancing, similarly, is a way to distribute any portion of the processing of requests between independent components of parallel processing systems (van der Boor et al., 2021;

Membey et al., 2012). Commonly load balancing is used to improve performance in queue lengths or delays (van der Boor et al., 2021). Load balancing by distributing tasks processing leads to minimizing the total execution time (Al Qayedi et al., 2015). For increasing availability and scalability, microservices require some load balancing tool (Kookarinrat and Temtanapat, 2016). According to Membey et al. (2012), load balancing is a broad concept: for example placing a database and a web server on different physical instances is already seen as load balancing. Traditional load balancing is commonly related to adding redundant servers to share the processing load of complex requests (Membey et al., 2012). Multiple redundant instances deployed on several physical systems support horizontal scaling and increase fault tolerance, enabling load-balancing (Richter et al., 2017). Load balancers help to address critical issues with the performance, economy, and availability, which is why load balancers continue to develop (DeJonghe, 2018). Load balancing helps the availability of the service; however, systems should have also redundancy of services to avoid a single point of failure - a single service that would stop the whole system if it becomes unavailable (Membey et al., 2012). In such a case decoupling the service communication using asynchronous messaging, queues can increase fault tolerance (Richter et al., 2017).

Performance optimization methods for microservice cloud systems are more complex than for monolithic architectures; however, the performance of one microservice certainly affects the others (Zeng et al., 2022). The systems' throughput is limited by the part with the longest event average processing time; optimizing the bottleneck can optimize the overall throughput (Nakadi Manual, Recipes). Compared with traditional monolithic applications, communication between microservices will bring higher latency (Wan et al., 2020). When chained service requests traverse more than one microservice, the latency of longer chains is longer than that of shorter chains (Wan et al., 2020). Microservice performance is affected by multiple layers, from software architecture and application design to the underlying hardware system (Zeng et al., 2022). To optimize cloud microservice-based applications, many researchers focus on solutions to reduce the latency of user requests (Wan et al., 2020). The latency of the user request in the microservice chain is mainly composed of three parts, network latency, queue time, and service time (Wan et al., 2020). The network latency refers to the time a user request spends in-network transmission; service time is the time user requests run in the application; the queuing latency refers to the time a user requests waiting in a message queue for execution (Wan et al., 2020). The waiting time in the message queue may significantly impact latency (Wan

et al., 2020). Suppose a request has extended the total queue time due to entering the message queue at multiple microservices that it traverses. In that case, the total latency will significantly affect the queue latency and service time (Wan et al., 2020).

2.4 Queueing theory

The queueing models have often been used for modeling computer systems and can be an effective performance analysis tool (Spies, 1996). Queueing theory can be used to model microservice latency to analyze performance (Wan et al., 2020). Researchers used queueing models in multiple studies, for example to develop an approach to evaluate and predict the performance of Kafka (Khazaei et al., 2012; Wu et al., 2019), to evaluate elastic cloud applications' load-balancing approaches (Salah and Boutaba, 2012), to propose load-balancing algorithms to find the minimal number of computing resources needed to meet the service-level objective requirements (Al Qayedi et al., 2015), to develop a performant scheduling scheme for MapReduce (Wan et al., 2013) and to model performance of Softwarized Network Services (Prados-Garzon et al., 2021). Experiments or simulations commonly verify the models. The studies found that queuing theory is applicable for analyzing or improving the analyzed system's performance. Queueing theory models help to estimate waiting time in the queue given arrival rate to the queue and service rate. It is a tool which helps to compare different queue models for effectiveness and to determine optimal number of workers serving the queue for certain waiting time requirements.

In queueing theory, the unit demanding service is called a customer, and a unit providing service server (Bhat, 2008, p. 1). Commonly queue input is assumed to fit the Poisson distribution, which is considered an appropriate model for events occurring at random (Bhat, 2008, p. 17). Several models represent various queues, servers, distribution of customers' arrival, and processing times.

One queueing system, the multiserver queue (usually denoted with M/M/s in the literature) is a system with single queue and multiple servers like depicted on the left of Figure 3. It is the model used most in analyzing queueing systems with multiple servers, where customers' arrival is assumed to occur at random (Poisson process), and service times are assumed to have an exponential distribution (Bhat, 2008, p. 43). In such system servers provide service independently from each other (Bhat, 2008, p. 43). Arriving customers form a single queue, with the first customer entering into service as soon as a server is accessible; hence, no server stays idle as long as there are customers to serve (Bhat, 2008, p. 43). For the discussion on customers' waiting times, it is assumed that they are served with a first-come, first-served queue

discipline (Bhat, 2008, p. 47).

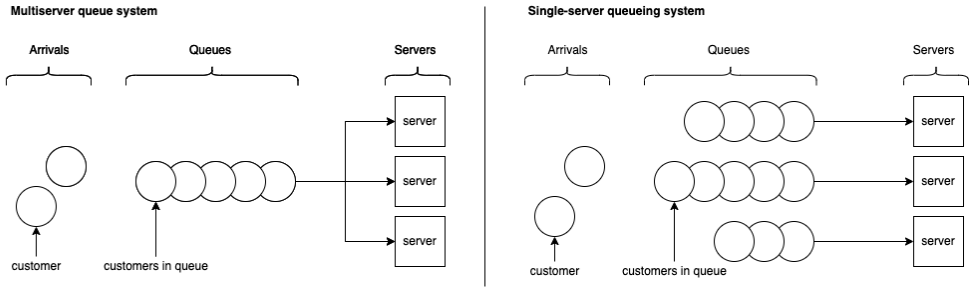

Figure 3: Queueing systems with multiple servers

Single-server queueing system (M/M/1) illustrated on the right of Figure 3 is a system in which queue has a single server. When multiserver queue and single-server queue models are compared for performance it is often concluded that multi-server queue is faster for customers. However, there are exceptions which depend on the modelled behaviour of customers joining single-server queues. For example Changfu and Zhenyu (2009) found that single-server queues might be as performant as multiserver queues, if an arriving customer always chooses the shortest queue to wait for services and can shift to the shorter queues at any time (Changfu and Zhenyu, 2009). A multiserver queuing system has been proved to have no difference from a single-server queueing system with such discipline (Changfu and Zhenyu, 2009). If the customer arrival streams are independent Poisson processes and service times are exponentially distributed, a multi-server queuing system is equivalent in performance to a single-server queuing system (Changfu and Zhenyu, 2009). The Join-the-Shortest-Queue policy achieves minimum mean delay among non-anticipating policies (van der Boor et al., 2021) in case of multiple queues. However, most of the studies model the situation like in (Cheng and Li, 2011) paper in the case where arriving customers join the shortest line, but switching between the queues is not permitted (Cheng and Li, 2011). The efficiency of multiserver queues over single-server systems is the reason why multiserver service systems, whenever possible, use single waiting lines feeding multiple counters for service (Bhat, 2008, p. 51). The time customers stay in the system is an essential indicator of the service quality of systems. Comparing service with two lines and service with one line Xiao and Zhang (2010) found that the "first-come, first-serve" principle is more time-effective or, technically, one queue is better than more queues in a system with multiple servers (Xiao and Zhang, 2010). To conclude, in most cases, a single queue with multiple servers is more efficient than multiple queues with a single server.

A system may be composed of more than one queueing stage. Queues in series, or tandem queues, are the simplest queueing network structure in which service facilities are located in series and customers pass through them sequentially (Bhat, 2008, p. 114). One example of such queues is sequential computations in a computer system. The system operates as a queue with one or more servers at every service part. Customers from outside the network always start at the first facility (Bhat, 2008, p. 145). There is no blocking between successive service stations; this means that the waiting rooms feeding customers to these stations have infinite capacity (Bhat, 2008, p. 145). In the case of the series of queues, we may conclude that even though in finite time, the individual queues are not independent, in the long term, they behave as if they are independent (Bhat, 2008, p. 146). So the queueing stages can be analyzed separately.

Customer inter-arrival times do not always follow an exponential distribution. Changfu and Zhenyu (2009) studied a more typical customer arrival situation with peak period distribution characteristics, and found that the queuing theory is still applicable for such processes (Changfu and Zhenyu, 2009).

3 Context and objectives

3.1 Connected Retail Inbound pipeline

Connected Retail (CR) is a system created at Zalando to enable brick-and-mortar stores to sell their products on the Zalando platform. Currently, around 7,000 active stores participate in the program (Retail). CR system consists of two parts: inbound, responsible for article data delivery to the Zalando platform, and outbound, responsible for Zalando customers' order-related services.

CR Inbound pipeline is developed to deliver physical retail stores' inventory information to the Zalando platform. It aims to simplify the process of inventory change propagation for retailers by providing REST API for file upload. Retailers are expected to send the whole inventory of the products at least once every day. In addition, retailers should send any product quantities or price changes as soon as possible. The CR system load is growing as new stores are added; hence the pipeline needs to be optimized to be able to handle even the intensive load periods.

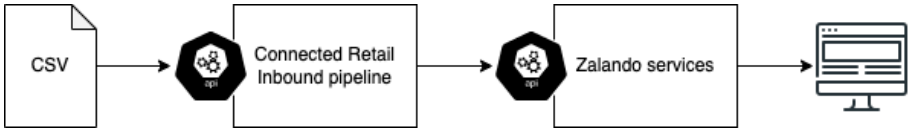

Figure 4: Connected Retail Inbound pipeline

The inbound pipeline contains services written in different programming languages and deployed to the AWS cloud in Kubernetes containers. Additionally, the system utilizes serverless computing components (AWS Lambda) and an AWS S3 as a storage system for files. The team develops applications as API-first microservices (Tschumak and Frauenstein, 2019). The pipeline performance is required to remain below defined Service-level objective (SLO). SLO response time for the pipeline is calculated from the initial state until the final state of the request processing (Al Qayedi et al., 2015).

The Connected Retail Inbound pipeline connects to the Zalando platform via stock and price APIs. APIs are rate-limited; that is why data of stock and price updates is cached within the pipeline to ensure as few updates are issued towards Zalando API as possible while assuring that the stock and price information is eventually consistent. In addition, inbound pipeline services use caches a circuit breaker pattern (Circuit Breaker) for calling the platform and other APIs to provide stability and avoid cascading failures when API call number exceeds the rate limits or services are unavailable.

Before modifications of the pipeline described in this thesis (original state), services inside the system are connected by standard AWS SQS queues which do not guarantee message ordering. In case of anomalies with queue message processing like service dependency unavailability, message rejection from downstream service, or problems with message format, dead letter queues are utilized to store unprocessed messages. Dead letter queues are monitored, and messages are manually sent to the main queue to be reprocessed if possible. Reprocessing the failed messages is necessary to ensure that all retailer updates are timely forwarded into the Zalando platform. On the other hand, the resurrection of the letters introduces updates out-of-order, which means that while processing revived messages together with regular updates application has to check data timestamps to avoid overwriting the latest data.

Additionally, services publish Nakadi events in case of essential operations like export of the changes in prices or stocks; however, the events are not consumed by any other services and serve mainly operational (debugging issues) and analytical purposes. Architecturally, the pipeline is built similarly to the *chain of responsibility* design pattern, where the next service consumes the output of the previous service processing. Most services have databases, use one or multiple queues for message input and output, and call other services synchronously via RESTful API.

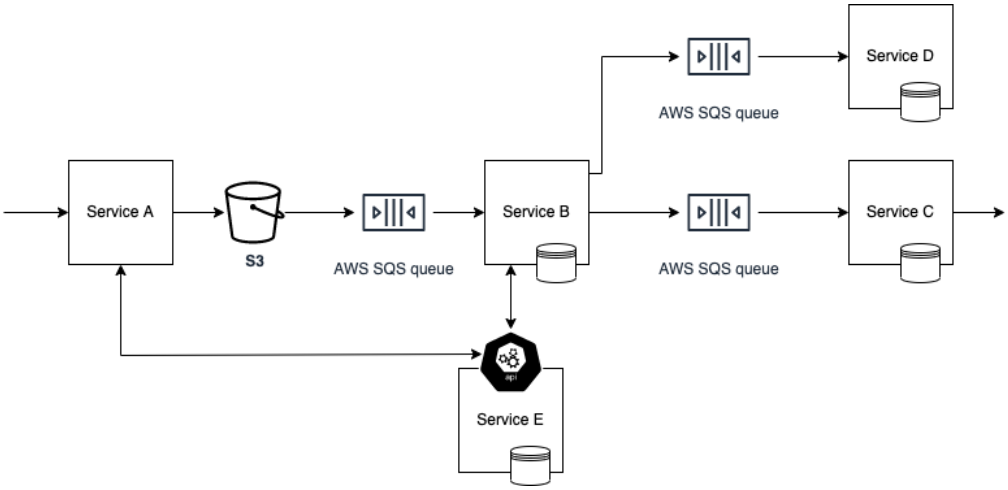

Figure 5: Connected Retail Inbound pipeline microservices system fragment, original architecture.

The simplified pipeline fragment in Figure 5 shows the part of the main flow of retailer updates processing. The main article updates flow includes Service A, B, and C. Additionally, the pipeline has a separate branch where article data is forwarded to Service D for later use by other components. Multiple pipeline services,

for example Service A and B, call REST API of configuration Service E. Service B is a legacy service that writes to two SQS queues. Service B also performs heavy database operations to save retailers' data to the database and to reset the stock of articles not encountered for a particular time. Service E is used by many services in CR inbound and outbound because it provides shared configurations. The hard dependency of multiple services on Service E, makes it a potential single point of failure. For example, Service A and B call it using a point-to-point communication model, which means services availability depends on the availability of Service E. Although the caches for calls are used, processing in the pipeline will be affected if Service E is unavailable for a prolonged period. The backlog of the messages in dead letter queues will increase, and updates will not reach the destination until the problem with Service E is fixed. All services pictured on the diagram use Kubernetes scalable deployments with multiple service instances (pods). In case of increased message count in input queues or increased Central Processing Unit (CPU) utilization of the application, services are horizontally scaled, i.e., additional pods are added to the service. Such scalability allows for dealing with increased load. Load from the retailers varies, as the diverse file sizes arrive at different times of the day depending on the number of updates. When a retailer sends an entire inventory at least once a day, the size of the files is much larger than the other price or stock update files.

3.2 Objectives

As described in previous chapters, communication patterns strongly influence microservices system architecture and performance. They strengthen or weaken the independence of services, contributing to system availability and reliability. There are numerous approaches to developing service interactions. This study aims to evaluate the impact of two communication-related methods on microservice system performance. The approaches are evaluated using queuing theory model and experiments on actual implementation. Modifications of the system in the focus of the thesis:

1. Ordering article messages to optimize processing in the main system flow
2. Introducing parallel process flow for less time-critical operations and removing synchronous API calls

Criteria for analyzing the approaches:

- performance in terms of throughput and latency of the system fragment
- fault tolerance and resilience
- complexity
- extensibility
- manual recovery possibility

The thesis aims to answer the research questions by testing assumptions on real live microservices systems and comparing the results with research papers, while documenting changes and their effect on performance. Performance is defined in terms of latency and throughput of the system fragment measured before and after the change on the affected components. Therefore, assuming that improvement in the part of the system will contribute to the whole system's performance improvement.

The following research questions are explored:

1 - Will the migration to ordered messaging enable optimizations in microservices and lead to microservices software system performance gain?

2 - Will introducing parallel process flow for less time-critical operations and removing synchronous API calls improve performance and reliability of the software system?

4 Microservice systems restructuring approaches

4.1 First approach - ordered messaging

The first approach of pipeline optimization, further referred to as an ordered messaging approach, was designed with the following assumption: the migration to ordered messaging will enable optimizations in microservices and lead to microservices software system performance gain.

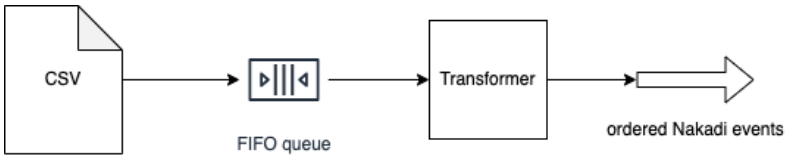

Figure 6: Initial design idea for ordered messaging approach

Data indicated that databases were a significant performance bottleneck; reducing the number of writes, a microservice performs to its database improved its latency and throughput. One of the reasons to maintain a database in the pipeline services is to avoid the possibility of overwriting new information with the previous data. It is crucial to avoid losing the latest data in the Connected Retail business context. For example, if the retailer has sent quantity 1 for an article and shortly after that quantity 0, the result in the Zalando platform should be 0. Otherwise, an order for the sold-out item might be placed, which leads to the cancellation of the order from the platform side and an unpleasant customer experience. To export data in the correct sequence, every data timestamp is compared with the previously written data to maintain consistency. Such a compensation mechanism is necessary as the standard AWS SQS queue, used as a messaging system between services, does not guarantee order. In case of short latency between updates, message position might be switched while inserting into the queue or processing in parallel by different instances in the previous service. Corner cases for race conditions of very close updates are still possible, but an approach with a database solves most of the issues. The system relies on eventual consistency, as retailers must provide complete inventory update daily; eventually, the prices and quantities return to the correct values.

Maintaining the order of the messages - stock and price updates - can eliminate the need to check the last update times in the database and, more importantly, removes the requirement to write to the database on every data change. Ordered messaging could help to increase performance in the pipeline. The initial design idea, visible in Figure 6, was to replace existing standard AWS SQS queue with

FIFO type queue for file upload notifications. Then parse the file extracting article information for stores using a new transformer component and publish article updates to the publish-subscribe message broker (Nakadi), preserving the order. AWS SQS FIFO queue for file notifications was chosen as it integrates the same way with the file storage S3 as previous queue type. Nakadi was selected for publishing article messages because it supports ordering events and multiple consumers without the need for duplication of queues and loose coupling between services. Moreover, the team had previous experience with technology when publishing messages for analytics. Another important consideration was the persistence of the messages provided by Nakadi and the ability to query Nakadi events in a company Data lake, which helps to resolve incidents or support issues. Nakadi communication leaves a trace of each message content, which SQS does not provide. In case of processing failure, messages can be replayed in the same order. Using Nakadi also allows for minimizing the variety of the technologies used in the team, which makes maintenance easier.

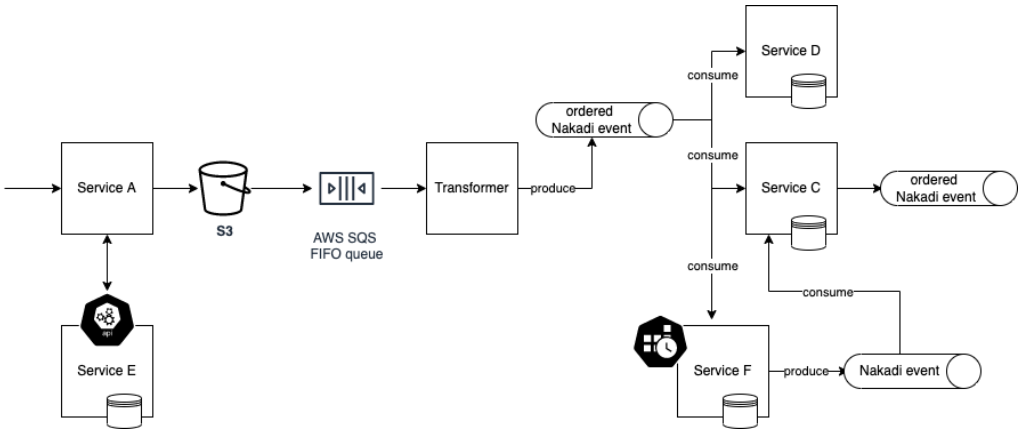

Figure 7: Connected Retail Inbound pipeline microservices system fragment, restructured with the ordered messaging approach.

Figure 7 visualizes service interactions with the ordered messaging approach. Similar to the system's original state, Service A pre-processes and validates the file sent by the retailer and uploads the result into S3 bucket. Validation service requires configurations provided by the Service E accessible via API. Uploading a Comma-Separated Values (CSV) file to an S3 bucket triggers messages inserted into the ordered queue instead of standard queue to preserve the ordering of updates. The transformer processes data from the S3, extracting article updates and pushing them to Nakadi partitions. The same partition is used for all events from the partner's store to preserve the order of messages. Produced Nakadi events are used by other services (Service C, D, F), which may rely on ordering the messages and don't have

to keep track of the message timestamp. Such ordered messaging design still has a problem in case of very close updates, as pre-processing service might publish to the cloud storage in the wrong order, resulting in the wrong order throughout the pipeline. Very close updates case also caused issues for the original implementation with an unordered message queue. The system in both cases uses the timestamp of message creation from the S3 bucket upload, which might create the wrong placement in the same situation. Eventually, with new updates from the partner, data will be again consistent.

4.2 Second approach - parallel process flow

Conceptually, the pipeline performs two operations: propagating changes sent by retailers and resetting stock according to business rules. Previously, both processes were performed by the services in the main pipeline. The idea is to offload the stock resetting functions outside the pipeline into a parallel process flow. A parallel process flow is defined as a flow executing one or more activities in parallel with the main flow. Parallel flow is introduced by determining which inventory should be reset as a side process outside the main flow, leaving only the actual change propagation to the main pipeline. Generally, the stock is reset on two occasions: when an article is not updated during the configured time or when article data does not pass validation against configuration.

Service responsible for resetting stock after configured time placed inside the main pipeline in the original state is removed. It is replaced with one of the components of parallel flow. The system in the original state conducts configuration-based validation as follows: validation component checks configuration; if the value sent by the retailer does not comply, the stock is reset for an article so that such product is not sold on the Zalando platform until it is corrected. Whenever an incorrect article arrives, a stock reset message is sent and processed in multiple services. The article information sent by retailers is persisted in the pipeline. Instead of processing stock reset every time a non-compliant value is encountered, reset can be performed on configuration change resetting all articles, not fitting configuration only once. Every time the retailer configuration is changed, the event is published. When the configuration change event is consumed, the database is checked for articles not complying with new settings. If such articles are found, their stock is reset. Such an approach will make configuration constraints propagate faster as stock resets. The validation stage can then skip all the incorrect data without resetting. If incorrect data is skipped before further processing, less invalid data will need to be processed. The load on the

pipeline will decrease as the amount of stock resets decreases from every time article is sent to once per article. The amount of processing inside the main pipeline decreases by performing stock resets outside the main flow and validation improvement. The parallel process flow approach differs from the ordered messaging approach by the communication remaining unordered within the main flow leveraging SQS queues. Additionally, the services previously using synchronous REST API calls start to use asynchronous event sourcing.

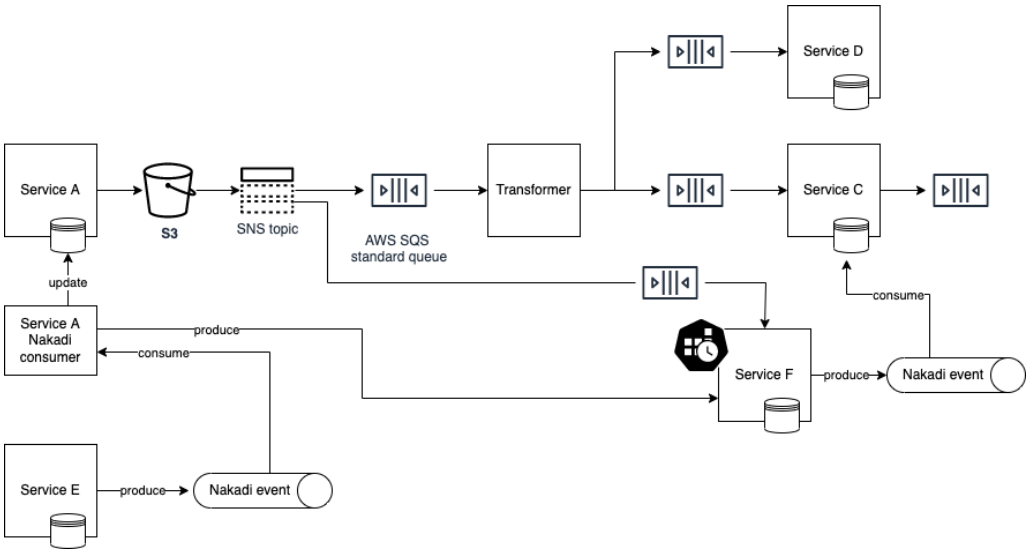

Figure 8: Connected Retail Inbound pipeline microservices system fragment, restructured with the parallel process flow approach.
Main processing flow: service A, transformer and Service C. Parallel process flow for stock reset: service A Nakadi consumer and Service F.

Service A needs several modifications to enable the decoupling of services A and E previously linked by synchronous API calls. The database replaces the cache. A database stores the copy of a subset of configurations from Service E needed for Service A. When the service updates the cache multiple times in an hour, the database is updated only in case it has no data for the requested retailer, which commonly happens only on the first call from the retailer. To keep database data up to date, a separately deployable component was implemented - Service A Nakadi consumer. Service A Nakadi consumer listens to Nakadi events produced by Service E on the configuration changes. This component would update the corresponding information if the configuration change was related to data stored in the Service A database. Such a solution speeds up the propagation of changes from Service E to Service A compared to caching solution because the cache provides outdated data during the caching period. Besides, database reads will presumably be faster

than calling service remote API. The other significant benefit of such a solution is the independence gain of Service A; it will be able to process most of the requests in the case of downtime of Service E. Although, there is still a dependency on the Nakadi events of Service E. If Service E fails to publish Nakadi events when configurations have changed, Service A might be processing requests with outdated settings. Luckily, such a situation is unlikely as Service E uses transactional event publishing. Configuration update and event publishing happen in one database transaction, which prevents updates without events. Besides updating the database, Service A Nakadi consumer publishes new ordered Nakadi events if the configuration stored in the Service A database is changed. The events are used for resetting the stock of articles by Service F.

Additionally, Service F runs cronjobs resetting stock for articles not sent by retailers during a configured time—this functionality requires expensive database queries previously performed in the main pipeline. Service F is now responsible for collecting the data, determining the resets, and publishing Nakadi events for the articles needing resets. Service F publishes the Nakadi event with resets consumed by Service C; this way, the data is pushed back to the main pipeline. Thus Service A Nakadi consumer and Service F form a separate flow of stock reset. Instead of propagating stock 0 for every invalid article from validation, articles are reset on configuration change and using general reset logic. A change should decrease the updates count in the pipeline. In addition, restructuring the system should reduce the processing latency because of the removal of Service B.

5 Evaluation

The original pipeline had several bottlenecks. A considerable number of write operations to databases in many services, a legacy service performing expensive database queries and hard dependency on configuration service availability. To address the issues two alterations to the original state were trialed: ordered messaging and parallel processing flow.

5.1 First approach - ordered messaging

The ordered messaging approach had the potential to substantially decrease database writes in pipeline services because preserving the order of the updates removes the need to save the latest state to the database. However, it became apparent throughout the implementation phase that the unbalanced partitioning of events would prevent the successful adoption of this approach. The described design was unusable for two reasons: quotas for AWS SQS FIFO type queues and slow processing of Nakadi events due to unbalanced partitions.

AWS ordered queues are restricted with a maximum of 20,000 inflight messages (AWS quotas). Inflight messages are the messages received from a queue by a consumer but not yet deleted from the queue (AWS quotas). If the quota is reached, the queue returns no error messages, but the processing stops because the messages are no longer returned to the consumer (Gregory, 2021). This limit might interrupt the processing of queue messages. Suppose the slow consumption queue accumulates more than 20k messages. In that case, the consumers which finished processing messages will not be able to get new data, but the producer will send more messages to the queue. In the end, there will be a queue with more than 20k messages and multiple idle consumers, as the messages will not be returned. S3 buckets provide only a few event notification options. The fact that AWS SQS FIFO queue has low quotas, made implementing ordered bucket notifications more challenging.

The second and main reason for abandoning the ordered messaging approach was performance degradation because of unbalanced partitions. The partition key was chosen as the hash of the partner store identifier. Such a partitioning strategy guarantees that all store messages are ordered, which is important for stock and price updates. This partitioning strategy also provides additional benefits like log compaction. If all messages of the same store and product are published to the same partition, in case of slow processing, there is a possibility to leverage log compaction - preserve only the most recent update for the article. However, in practice, it

turned out that store-based partitioning results in the highly unbalanced partitions. The limitation that one partition can be consumed only by one consumer in the consumer group means that consumers receive an unequal load. One consumer per one partition constraint leaves no possibility of load balancing or load sharing, which makes overall message processing slower. Latency increased during the load test due to a throughput decrease, as some instances could not cope with a large load due to overloaded partitions. Theoretically, the concentration of load in certain partitions might be solved by redistributing messages to the partitions with a shorter lag as described by Hunter et al. (2021). Unfortunately, redistribution can lead to the processing of messages out of order, which ordered messaging aims to eliminate. As visible in figure 7 Service C produces new ordered Nakadi events as an outcome of processing. The article update processing does not end in the fragment shown in Figure 7, but includes more service components with additional event consumption. In the worst-case scenario, the same message might be published to the partitions with the largest lag multiple times in the pipeline, which results in multiplied delay and larger differences in update processing speed.

Over-partitioning of Nakadi events will not entirely protect partitions from the hotspots and might increase the probability of idle consumers. In case the load is concentrated in one partition, even if there are more partitions than consumers, the partition load is targeted only onto one instance. The hotspot might remain if the system is scaled up until every partition is consumed by one instance. An overloaded partitions make event handling slower as using multiple instances for processing one partition load is impossible. If the consumers are scaled to match the partition number, more resources will be needed, but actual processing latency will still be suboptimal. Load distribution is more optimal with equal partitions than over-partitioning with a concentrated load.

The input sizes of inputs from different retailers are partly unpredictable. To evenly partition data from stock feed input requires load prediction. The simple prediction model of estimating a similar daily load will not work in this case because retailers can unpredictably onboard more stores. Other retailers may terminate stores, or stores might become inactive because of technical problems. Store activity also depends on the orders because the store might be set inactive when reaching the configured amount of orders. New retailers and stores are added. Developing such a prediction model would require considerable effort and would not have a large enough impact. From a practical point of view, if partitions are almost equal at one point in time, adding a new store with a large assortment into the system

will mean that partitions will become unequal as all store products will be updated through one partition. When all new store updates are placed into the partition, it will have an excessive load compared to other partitions overloading the consumer. Using over-partitioning of the topic would be beneficial if the load is inserted into a partition with less load and a new instance is added to consume it. Controlling such behavior would introduce overhead. Assigning stores to partitions might introduce timing issues; for example, if stores are assigned to the same partition, change the update sending behavior so that the updates will arrive closely in one period. For a consumer of the partition, it would mean the inconsistent load on different periods, from idle to extreme. As changing partitions—re-partitioning—on run-time is not supported in Nakadi, and such problems will inevitably occur in the long term.

Queueing theory is used to model differences between the original SQS approach and Nakadi partitioned messaging. Messages play the customer's role in a model when service instances are servers. Supposing that messages arrive at random following the Poisson process and processing times of servers form exponential distribution, the models of multiserver and single-server queues can be used to analyze the time messages wait for processing. SQS message processing is modeled as a multiserver queue, as one SQS queue is consumed by several server instances sharing messages. Nakadi topic is modeled as several queues - a queue for each partition, where each partition is consumed by one server, hence the single-server queue model applies.

For analysis, the following numbers were extracted from monitoring tools for numerical comparison of the queue systems. The average processing of a single article update in Service C is 69.5 ms. The message contains up to 10 article updates, so let the estimated message average processing time be 695 ms. This means that per second service can process 1.44 messages. According to SQS statistics, 9.89 messages are added to the queue every second. Service uses load balancing; on average, 8 nodes are running in the system during high load.

The waiting time in the queue for single queue and multiple servers according to (Bhat, 2008, p. 48) is calculated as follows:

Arrival rate: $\lambda = 9.89$; service rate: $\mu = 1.44$, number of servers (instances): s = 8 then $\rho = \frac{\lambda}{\mu * s} = 0.858507$

P_0 - probability for 0 messages in the system:

$P_0 = \left\{ \left[\sum_{i=0}^{s-1} \frac{(s*\rho)^i}{i!} \right] + \left[(s\rho)^s * \frac{1}{s!} * \frac{1}{1-\rho} \right] \right\}^{-1}$; $P_0 = \mathbf{0.000684}$

L_q - average number of messages in queue: $L_q = \left[\frac{1}{(s-1)!} * (\frac{\lambda}{\mu})^s * \frac{\lambda*\mu}{(s\mu-\lambda)^2} \right] * P_0$; $L_q = \mathbf{3.60147}$

L_s - the average number of messages in the system: $L_s = L_q + \frac{\lambda}{\mu}$; $L_s = \mathbf{10.469526}$
W_q - waiting time in queue: $W_q = \frac{L_q}{\lambda}$; $W_q = \mathbf{0.364153}$ seconds
W_s - waiting time in the system: $W_s = \frac{L_s}{\lambda}$; $W_s = \mathbf{1.058597}$ seconds

The waiting time in the queue for a single queue with a single-server model is calculated as: $W_q = \frac{\lambda}{\mu(\mu-\lambda)}$ (Bhat, 2008, p. 37) where λ - is a arrival rate of article updates, μ - is a service rate - rate with which service processes the updates. Article update in this case is one message with up to 10 articles. If instead we split the load equally across 8 partitions, one partition per service instance we get: $\lambda = \frac{9{,}89}{8} = 1,236\ arrivals/second; \mu = 1.44\ messages/second$; result is $W_q = \frac{1{,}236}{1.44(1.44-1{,}236)} = \mathbf{4,21}\ seconds$.

From the calculations, it is visible that the waiting time of the message in the queue increases when the queue is split into several separate queues. This explains why the SQS performance of one queue with multiple consumers was better than Nakadi's ordered messaging with partitioning - a single-server queuing system. As arrival rate was calculated by dividing across partitions equally, calculations show the case where partitions receive equal load, which corresponds to the random partitioning in Nakadi unordered messaging. In the case of ordered messages, the partitioning follows a hash strategy where the unbalanced partition probability is higher, which might introduce idle partitions which will increase overall message processing.

If the same load is divided into 7 partitions instead of 8 as occurs if one consumer is left idle, the waiting time in the queue increases multifold. With arrival rate $\lambda = \frac{9{,}89}{7} = 1,41\ arrivals/second$; and same service rate as previously, the waiting time in the queue is $W_q == \mathbf{32,64}\ seconds$. The difference between single-queue single-server calculation results for 7 and 8 partitions shows the adverse effect of unbalanced partitioning with idle consumers. To achieve similar performance to SQS queue with equal message partitioning, 20 Nakadi partitions are needed. The calculations do not consider the potential improvement of service rate due to the removal of the database operations. To be equal to multiserver queue performance, the service rate for single-queue single-server with 8 partitions should be increased from 1.44 to 2.56, removal of the database operations has potential of improving service rate only to the value around 2.

Figure 9 illustrates three different systems models: a single queue multiserver system corresponding to SQS queue setup, a single-server queuing system approximating partitioned Nakadi events, and a single-server for multiple queues as over

partitioned Nakadi event. Previous calculations compare SQS queueing system 5.1 and partitioned messaging system, where each partition is served by a single consumer analyzed in calculation 5.1. As shown previously, the single queue system is faster than multiple systems with a line per server. Even if over-partitioning the topic can help to spread the load to the consumers more equally, reducing the number of overloaded consumers, it will not improve the overall speed more than in calculation 5.1, as the calculation analyses the load divided equally. On the contrary, according to Lapadula and Levy (1996) in the multi-queue single-server model, overhead can be introduced by switching between the queues, and waiting time in the queue increases if the customer serving does not follow the first-come-first-serve principle.

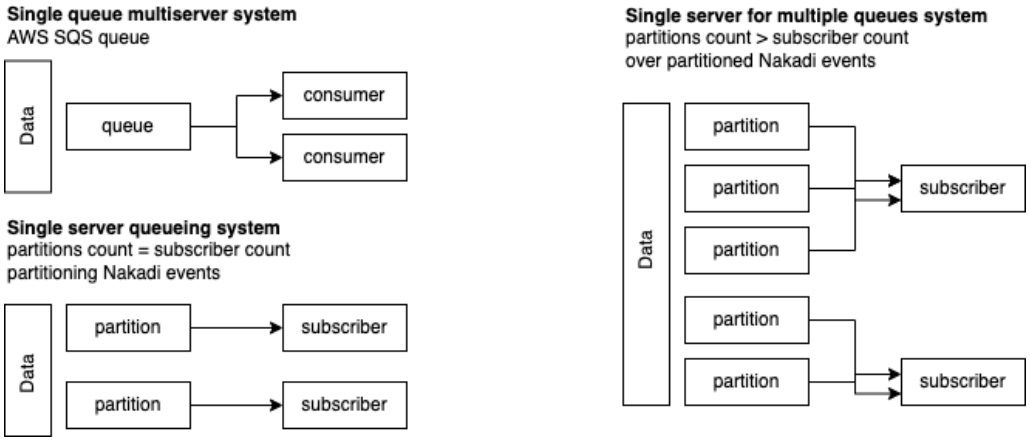

Figure 9: A single queue multi-server system corresponding to SQS queue, a single-server queuing system approximating partitioned Nakadi events, and a single-server for multiple queues as over-partitioned Nakadi event.

To conclude, the performance of the processing stage in terms of throughput and latency of the system fragment was not improved using the ordered messaging approach due to unbalanced partitions performance degradation. Fault tolerance and resilience decrease as the scaling of instances is coupled to partitions count; failure to process one message may block all messages in the partition. The solution decreases the system's complexity, as one Nakadi topic is consumed by different services instead of multiple queues. Besides, possible optimizations like database removal would make the system less complex. The solution is extensible due to the possibility of adding more consumer groups to the Nakadi topic. Nakadi messaging provides a manual recovery possibility, as the events are persisted and can be reprocessed.

5.2 Second approach - parallel process flow

Parallel process flow was implemented in several stages. First, replacing the REST API calls were replaced with database reads and consuming configuration change events is introduced to keep the database synchronic with configuration service changes. Service F is implemented for time-based and validation-based resets. Next, configuration change events enabled configuration-based resetting of non-compliant articles and filtering of all invalid values during pre-processing in Service A. Finally, Service B was deprecated with the replacement of the file to article message transformer.

The first improvement implemented to offload logic from the main flow is to replace cached point-to-point service calls with fetching data from the database in Service A. The change decreased synchronous calls to configuration service to less than 0.1% of their initial number. However, the configuration service API calls number did not drop to 0 because, in the case of new retailers, missing configuration data needs to be populated by a service call. Figure 10 shows the impact on the latency of Service A after deployment of the change. It is visible that immediately after deployment, 99.9th-percentile latency was unstable for a while, as calls to REST API were made to populate the database. However, later latency stabilized and, on average worst processing time decreased while throughput increased.

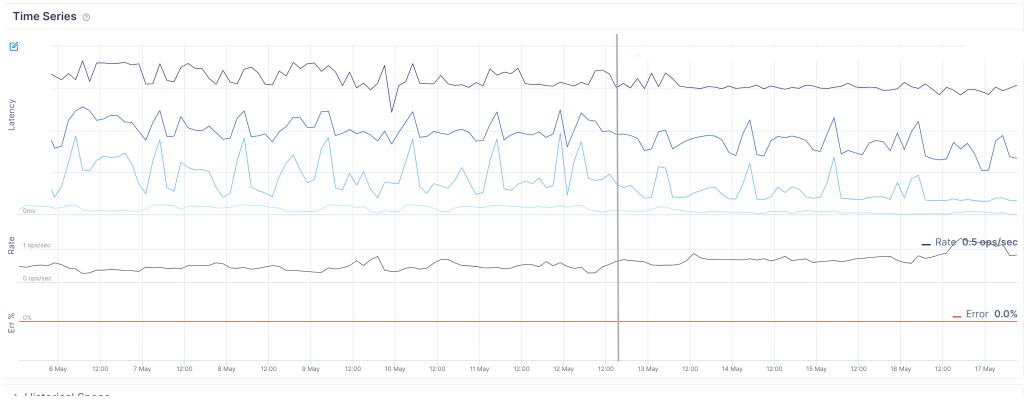

Figure 10: Service A latency with synchronous call to REST API endpoint versus fetching configuration from own database. The deployment time of the change is marked with a grey line.

To propagate file upload events to multiple queues, initially AWS *Lambda* component was used, later migrated to AWS Simple Notification Service (SNS) which can publish to several queues. The change decreased the average duration of message processing in the function, as visible in Figure 11.

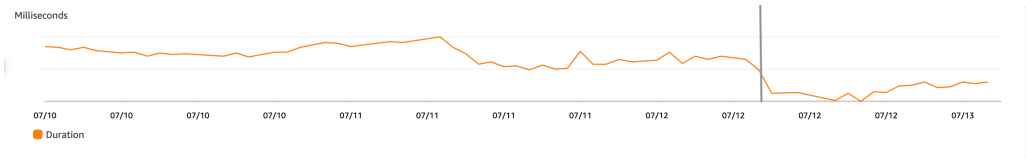

Figure 11: AWS *Lambda* processing duration before and after migration from AWS *Lambda* to AWS SNS to propagate file upload events to multiple queues. The deployment time of the change is marked with a grey line.

	Arrival rate	Service rate	Number of servers	Waiting time in queue	Waiting time in system
-	λ	μ	s	W_q	W_s
Service B	0.99	0.206	12	0.0029	4.8573
Transformer	0.99	0.588*	12	0.0000	1.7006
Transformer	0.99	0.588	6	0.0032	1.7039

Table 1: Queueing theory calculation for single-queue multiserver system comparing legacy Service B and transformer service rate impact on the waiting time.
*The service rate for Transformer 12 instances might be higher due to faster processing in the scaled component, for simplicity, it is assumed to be the same as for 6 instances.

Offloading the functionality outside the main flow into parallel process flow was gradually implemented. First, a new Service F for resetting the stock was placed outside the main flow; it consumed messages of the file uploads from a parallel queue. When the functionality of the resets was verified to work, reset functionality was removed from Service B. Next, a transformer to substitute the transformation from the file to SQS message previously done in Service B had to be implemented. Transformer implementation encountered issues with too fast writing to the queue, which was solved by modifications in a processing stream. The main challenge of the transformer was to implement reliable multiple message publishing from files. SQS queue supports only batches of 10 messages when a file can contain thousands of article updates. If one SQS update fails on all attempts, the service must retry the whole file.

According to queueing theory, an improved service rate decreases waiting time in the system and improves queueing system utilisation (Bhat, 2008). On average, the latency of the file-to-message transformer was 3.94 times faster than Service B. Measurements of service rates were taken from the same load processing by Service B with 12 instances and transformer with 6 instances. Queueing theory calculation comparing the queueing system with legacy Service B latency versus transformer

latency was performed using multiserver queue formulas from previous estimates in 5.1. As shown in Table 1, the waiting time in the queue of transformer service with six instances is comparable to the waiting time of Service B queue with 12 instances. Waiting time in a system with a transformer surpasses the same indicator for queueing system with Service B. Besides, waiting time in the system does not change rapidly, with decreasing number of instances by half. Results show potential in the improvement of pipeline fragment overall latency and throughput.

Service E, on its configuration change, produces events consumed in Service A Nakadi consumer. If an event affects data in the Service A database, the database entry is updated, and a new configuration change event is published. Configuration-based resets were implemented based on events produced by Service A Nakadi consumer component. Ordered messaging is used for configuration-based events, with events partitioning by the retailer. Due to the small number of events, no issues were caused by the unbalanced partitioning. One instance can handle all configuration change events. The resets on the configuration event are delayed for a short period to avoid the possibility of race conditions. A race condition might occur if Service A has already prepared the output file without considering a new configuration update. Suppose the transformer did not yet finish the file processing, but the stock resetting service gets a configuration change event. In that case, stock reset might be overridden by article updates. To delay events SQS queue with delay is used, as currently the amount of the configuration changes are small, the consistency due to parallelism is not an issue. The solution might need to be extended in the future to accommodate more concurrency.

During the development of the approach, the services became more modular. Modules performing different tasks are deployed as separate Kubernetes deployments. However, the code is partly shared and stored in the same repository. The increased number of smaller components requires more maintenance and troubleshooting, increasing system complexity. Modules of the same service share database, which may lead to database connection pool exhaustion if modules are scaled extensively on a high load.

The number of files output for further processing decreased by 0.87% after discarding all invalid rows. The change did not noticeably influence the latency of processing inside Service A. However total load in the pipeline decreased by 2.3% a day. As the pipeline is implemented as a chain, discarding some input in the earliest stages of processing means all the further processing will have less data, which should positively influence the performance.

The overall improvement of offloading logic outside the main flow is visible when Service B is substituted by a new transformer service in Figure 12 and 13. When the throughput of the transformer and Service B are similar, the lower latency of transformer contributes visibly to the 100th-percentile of total pipeline latency. As visible on the graph total processing in the pipeline spikes occasionally in both cases crossing the green horizontal line. The spikes are caused mainly by the larger number of updates and rate-limiting on the Zalando API. Time of processing the message in transformer compared to legacy Service B as evaluated previously is reduced by a factor of 3.94, which contributes to the average update processing time lowering. The pipeline's average total latency was reduced by a factor of 2.15 times with similar throughput.

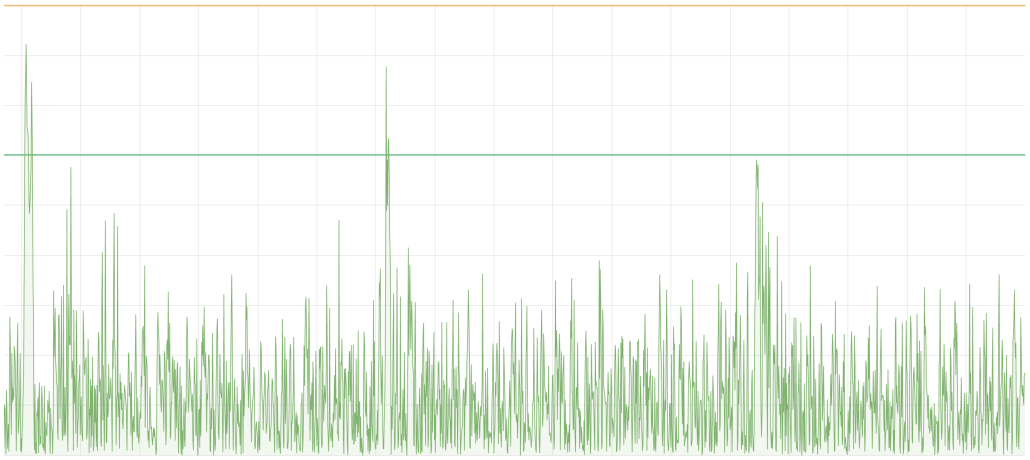

Figure 12: Total processing latency of the main pipeline **before** replacement of Service B with message transformer. The horizontal orange line marks SLO. Graph shows latency of pipeline processing on the Y-axis and time on X-axis.

To conclude, the performance of the processing stage in terms of throughput and latency of the system fragment improved using a parallel flow approach. Multiple factors contributed to improvement: improvement of message processing performance inside Service A, better efficiency of transformer compared to Service B, and deprecation of legacy Service B. Service A performance improved after changing REST API call to database reads, and discarding all invalid rows during the validation. The performance of the transformer exceeded the performance of Service B because the transformer does not use a database or point-to-point communication. The parallel process flow resulted in fewer updates in the main flow due to improved validation and preparation of stock resets in parallel to updates processing. The solution increased the system's complexity by introducing more application modules

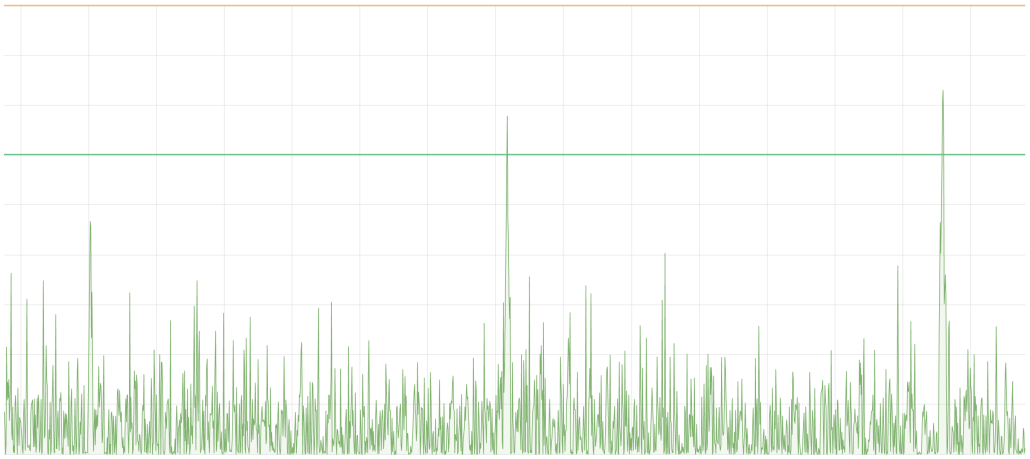

Figure 13: Total processing latency of the main pipeline **after** replacement of Service B with message transformer. The horizontal orange line marks SLO. Graph shows latency of pipeline processing on the Y-axis and time on X-axis.

and queues. The solution is extensible; as configuration changes Nakadi consumer can similarly consume changes from other services. In addition, the consumer can process additional configurations if needed, and multiple services can consume the same configuration change message in parallel. The parallel flow produces Nakadi events, which other services can consume. SQS queues preserved in the main flow provide the possibility of recovery from errors by reprocessing dead messages. However, primary flow communication with queues is not persisted. In the case of wrongly processed messages due to a problem, reprocessing and identifying such events might be more challenging than when using Nakadi messaging.

6 Discussion

6.1 Microservice independence and system complexity

Both evaluated communication-based design approaches attempted to enhance the independence of the services. The ordered messaging method used a publish-subscribe communication model instead of queues to decouple the number of published message duplicates from the number of consumers. This way, other consumer groups can quickly join the event processing. The service does not need to publish separately to multiple queues or possess any logic specific to consumers, like crafting particular messages for different service queues. Using a publish-subscribe communication pattern supports availability and reliability, changes in any aspect of publisher and subscriber services as far as the publisher can produce and subscribers consume messages in the same format as previously (Kul and Sayar, 2021). The parallel process flow approach strengthens the independence of services by switching from synchronous point-to-point communication to event sourcing for configuration updates. In addition, larger applications are split into independently scalable modules—for example, Service B was replaced by a different stock reset component—Service F, and the file-to-message transformer. On the one hand, splitting of the components increases the scaling independence. On the other hand, with an increase in the number of nodes in the system, operational complexity increases (Kookarinrat and Temtanapat, 2016). Although following the design principles of single responsibility, small service size and scalability are essential for microservices as suggested by Unlu et al. (2021); Megargel et al. (2021). In addition, changing REST API call between services A and E to configuration change propagation by Nakadi events increased tolerance of the failure of services. For example, service call failures might be caused by the unavailability of the infrastructure (Bakshi, 2017).

The system's complexity has decreased with the ordered messaging method due to fewer infrastructure components—one Nakadi topic instead of several SQS queues. Besides, reduced use of databases would simplify overall system complexity compared to the original state. On the contrary, the parallel process flow approach added the system infrastructural components by introducing additional SQS queue and SNS topic. A database replaced the cache, service and multiple modules were added when removing only one service. Complexity is the main disadvantage of the microservices approach (Richter et al., 2017). Unfortunately, increasing independence, resilience and scalability often increases complexity. The autonomy is more important as if the service independence cannot be achieved, the advantages of using microservices

are eliminated (Karabey Aksakalli et al., 2021). Complexity in the parallel flow approach was introduced to compensate for removing point-to-point communication. The drawbacks of point-to-point communication are degrading performance and increased number of development, operation, and maintenance (Karabey Aksakalli et al., 2021). Moreover, synchronous requests might cause cascading system failures if the service depends on the data or functionality of the called service (Janssen, 2017; Karabey Aksakalli et al., 2021).

6.2 Communication

Both modification approaches were based on event-driven communication patterns, which, can be implemented with the message queues and publisher-subscriber messaging infrastructures (Indrasiri and Siriwardena, 2018). Although in the parallel flow approach, the processing of the messages was branched into two flows, requests are still processed sequentially according to a *chain of responsibility* design pattern where the output from one service becomes an input for the other one (Akbulut and Perros, 2019) like in the original state and ordered messaging approaches. One database per service data storage pattern (Taibi et al., 2018) was used, except for multi-component services with multiple separately deployed modules accessing the same database. Sharing a database between modules in the long term might prevent modular components from scaling as the number of connections to the database is limited. The choreography communication design pattern was preserved from the original state through all the variants (Megargel et al., 2021), which maintains the autonomy of the services not relying on a central controller.

Asynchronous communication patterns prevailed in the central processing flow from the original to the modified system pipeline. The ordering approach changed the nature of asynchronous communication using the publish-subscribe model and ordering messages, and parallel flow approach helped eliminate synchronous API calls removing hard dependencies between services. Due to asynchronous multi-step processing within the pipeline, when the retailer uploads the file to the API, the response means that the data processing started rather than completed (Pacheco, 2018). This way, request-response communication is absent in the central processing flow on every stage except pre-processing.

The parallel process flow approach allowed Service A to become closer to *purist microservice*; it no longer depends on configuration service to perform its tasks; however, it is still producing a response to the caller service. Pacheco (2018) describes a *purist microservice* as a microservice that performs its task without the need to return

a response and functions without requiring communication with other microservices. Instead of depending on Service E API Service A depends on the database, which may have unavailability because of fail-over or maintenance downtime. Retries mitigate short periods of database unavailability. However, the new database in the pipeline adds to operational burden. In this light, the transformer component implemented to replace Service B fits the *purist microservice* description. It consumes file SQS message and produces article update messages and Nakadi events without communicating to any other services or returning any responses. Transformer differences compared to previous Service B manifest in the absence of dependencies: Service B had a hard reliance on Service E and database operations overhead. Transformer independence brings more flexibility, as independent processing of messages allows microservice architecture solutions to change over time (Whitesell et al., 2022).

Communication models used in both pipeline transformation approaches were broker-based, which enabled independent processing, buffering messages, and scaling communication (Whitesell et al., 2022). The parallel flow approach mitigates the effects of downtime in Service E, because the tight coupling of the synchronous call to Service E is replaced by Nakadi event consumption. Buffering the messages helps to deliver them when consumers can process them again (Whitesell et al., 2022). For large message numbers, broker-based communication performs more stably when the number of users increases, whereas the RESTful method may produce error responses (Hong et al., 2018)

The use of Nakadi for messaging has introduced the persistence of inter-service communication, which follows the best practices for microservices (Ntentos et al., 2021). Configuration change events, introduced in the parallel flow approach, use ordered communication, which is designed in event sourcing style, where state-changing events are stored in sequential order (Indrasiri and Siriwardena, 2018). Subscribers can restore the entity's state by processing the sequence of events for the entity (Indrasiri and Siriwardena, 2018). The ability to replay data will be essential if a bug is introduced and not noticed immediately. For example, resetting stocks in Service F uses Nakadi events to publish to Service C. In the early stages of adopting the new stock reset, there was a bug of exporting wrong prices with stock reset. The messages' persistence helped quickly identify which products and retailers were affected by the bug and resend corrected messages.

According to Oliveira Rocha (2022) with end-to-end partitioning - ordering the events throughout the pipeline, better performance can be achieved while avoiding concurrency and ordering problems. In the case of the ordered messaging approach,

the attempt to order events in the pipeline failed because of the unbalanced partitions performance issues. A partitioning strategy on the store level was chosen. Another option would be to partition events both by store and product. This way, updates from a new store addition can be spread across existing partitions instead of adding all updates to the same partition like in store-based partitioning. In practice hash of the store and product divided by a partition might not result in equally spread values, which will mean a greater load for some partitions. Hash partitioning results in unequal work shares because of hashing collisions (Korab, 2015) for a small set of keys. With a small number of partitions, the load might be so different that some consumers might be overloaded with messages, and some consumers might experience starvation (Korab, 2015). As hashing does not account for current events count in the partition, it provides worse distribution than joining the shortest queue model. There is, unfortunately, no guarantee of balanced partitions in this case, and even with balanced partitioning, the risk of temporarily overloaded partitions remains. According to Oliveira Rocha (2022) the partitioning key choice influence the load equity; however, there is still a possibility for temporal hot spotting. Products sold by retailers change every season, which means a balance of partitions may change unpredictably. Possible improvement for the ordered messaging approach is exploring possibilities of different key-based partitioning and over-partitioning the topic. Because increasing the Kafka partition count can significantly improve its throughput and improve performance (Dobbelaere and Esmaili, 2017; Wu et al., 2019).

In both the ordered messaging and parallel process flow approaches, when Nakadi messaging was used, the services implemented a process-first strategy for consuming the messages. The offset is committed only after processing the event, which might lead to a blocked pipeline; on the other hand, it guarantees that the updates are fully processed, and no updates are silently discarded (Fahland, 2019). Such a processing strategy is important in order not to skip any crucial updates from the partner. But might cause slowness in processing if there is an invalid message because it delays all the next messages. SQS queue differently from Nakadi delays only problematic messages, which might be beneficial for overall performance in terms of message processing latency.

Reprocessing failed messages causes duplicates in the transformer output queues. Duplicates do not cause issues in queues because the services after the transformer are built to handle them; previously, duplicates were also possible in the original state. However, the challenge of developing the transformer service was to reduce the

possibility of failed message processing with retry. Transformer service input can be quite a large file which during transformation is split into SQS messages with small batches of article updates. If, during processing, one of the SQS messages fails to be published the whole file will need to be reprocessed. It introduces additional latency for processing but also duplicates of messages which will need to be processed in the next service. Such failures may cause slowness in the pipeline. So the balance between single message retries and slow message processing versus retry for the whole file.

6.3 Performance

The simplest method to assess the success of microservices is to compare their performance with the performance of an equivalent monolithic implementation; the performance of the new microservice-based system should meet or exceed the pre-migration performance (Akbulut and Perros, 2019). Performance measures from monitoring tools of the service modules and AWS queues were used for comparison. Tracing tools give an overview of the application performance and allow comparing the service's average processing time with the same load during the load test. However, load test latencies in the production-like environment might show an exaggerated difference in performance, as the load test concentrates load in a smaller interval of time when the load is less concentrated in production. Running systems in parallel in production shows the actual performance difference. Production data is used to compare Transformer and Service B performances, as two different services consumed messages from 2 queues from the same S3 bucket source. The same service performance must be compared when the change is deployed to production. Latency and throughput change show performance change to some extent; however, the messages in seemingly equal load might need different processing, which introduces uncertainty of the change influence. For example, suppose the service conducts pre-processing of data-consuming files. In that case, each message can represent different files, and processing speed is correlated to the total file sizes more than the count of the files. In this case, evaluating performance with a controlled production-like load during a load test might provide better insights than production performance monitoring. However, performance difference usually is verifiable in both methods.

This study described two possible approaches for enhancing the performance of microservice systems by restructuring: ordered messaging and parallel process flow. The work is limited to the trial of the approaches to only one microservice system. When theoretical sources suggested that both techniques would increase

the performance, the practical implementation of the ordered messaging approach had an adverse effect on the pipeline message processing performance in terms of throughput. The parallel process flow approach improved the pipeline performance in terms of the worst message processing latency. Queueing theory-based assessment helped to evaluate the reasons for observed tendencies.

Finally, to answer the research questions, the following conclusions can be made:

1 - The results show that the migration to ordered messaging might enable optimizations in microservices but did not lead to microservices software system performance gain in the studied case.

2 - Introducing parallel process flow for less time-critical operations and removing synchronous API calls improved the performance and reliability of the software system.

6.4 Queueing theory

The waiting time in the message queue may significantly impact latency (Wan et al., 2020). For example, suppose a request has extended the total queue time due to entering the message queue at multiple microservices that it traverses (Wan et al., 2020). In that case, the queue latency and service time will significantly affect the total latency (Wan et al., 2020). Waiting time in the queue was modeled using queueing theory to analyze ordered messaging and the parallel process flow approach. For the ordered messaging method queueing theory-based waiting time calculations explain the performance degradation when using ordered Nakadi messaging by the more efficient message processing in the multi-server model compared to the single-server model. For the parallel flow process, the queueing theory was used to estimate differences in waiting times of the same model with different service rates. Increased service rate can decrease waiting time in the queue even with the smaller number of servers. Better latency performance can be achieved by running fewer instances for the service. Queueing theory provides insightful results of the changes to the microservice processing stage. It allows comparing time spent in the queue and the system at every step. Queues of the system were analyzed independently, as in the case of the series of lines; in the long term, queues behave as if they are independent (Bhat, 2008, p. 146).

6.5 Further improvements

Theoretically, the performance of joining the shortest queue model with message shifting from hotspot partitions with unordered Nakadi messaging should be as good as that of unordered SQS queues performance, because single-server queues are as performant as multiserver queues if an arriving customer always chooses the shortest queue to wait for services and can shift to the shorter queues at any time (Changfu and Zhenyu, 2009). The parallel flow approach can be further evolved to use such Nakadi messaging to obtain the communication persistence benefit.

As in the parallel process flow, messages for stock reset have lower priority than stock updates from partners, to distinguish requests from different microservice chains, multi-priority message queues could be used (Wan et al., 2020) as a further improvement of the approach. Priority of messages can speed up important updates propagation, improving parallel process flow design. Multi-priority queues could be used at the point where two flows join before Service C.

6.6 Limitations

One of the limitations of data used for queueing model calculations is that AWS monitoring provides only estimates not precise numbers of the messages number sent to the queue and oldest message age. Service monitoring tools provide more precise data. However, the processing of the article updates is monitored separately, not by the message; that is why the average article update time was taken and multiplied by the most common message size of 10 updates; however, smaller messages are possible is the case if the number of updates from the file is not dividable by 10.

Queuing theory has separate models for batch arrival; in this study, the message is considered a customer, but it could also be represented as a batch of customers arriving simultaneously; this would allow calculating the delay of a single article update. Besides, calculations did not consider possible latency between the SQS queue or Nakadi partition and service. Adding to the queue or partition might take different times depending on the network latency between service and message-broker.

7 Conclusion

This study described two approaches for enhancing the performance of microservice systems by restructuring: ordered messaging and parallel process flow. The work trialed these approaches on the Connected Retail inbound pipeline. When theoretical sources suggested that both techniques would increase the performance, the practical implementation of the ordered messaging approach had an adverse effect on the pipeline message processing performance in terms of throughput. The parallel process flow approach improved the pipeline performance in terms of the average worst message processing latency. Queueing theory-based assessment helped to evaluate the reasons for observed results.

Both approaches can be further improved. The ordered messaging approach might benefit from an increased partition count and better-chosen partitioning keys. System restructured by parallel process flow can be further developed by utilizing priority queues for messages with different criticality and employing a way for inter-service message persistence. The pipeline, in general, could be further developed by parallelizing different updates, for example, separating price and stock update flow.

The study contributes ideas of microservice restructuring approaches which might inspire practitioners. Encouraging to numerically compare communication approaches using queueing theory, which will help analyze microservice communication methods in further engineering work.

References

Akhan Akbulut and Harry G. Perros. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing*, 23(6):19–27, 2019. ISSN 19410131. doi: 10.1109/MIC.2019.2951094.

Fatima Al Qayedi, Khaled Salah, and M. Jamal Zemerly. Queuing theory algorithm to find the minimal number of VMs to satisfy SLO response time. *2015 International Conference on Information and Communication Technology Research, ICTRC 2015*, pages 64–67, 2015. doi: 10.1109/ICTRC.2015.7156422.

Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Bboris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang. Data infrastructure at LinkedIn. *Proceedings - International Conference on Data Engineering*, pages 1370–1381, 2012. ISSN 10844627. doi: 10.1109/ICDE.2012.147.

AWS quotas. Quotas related to queues - Amazon Simple Queue Service. URL https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/quotas-queues.html.

AWS SQS. AWS Simple Queue Service (SQS). URL https://nakadi.io/manual.html#aws-simple-queue-service--sqs-.

Kapil Bakshi. Microservices-based software architecture and approaches. *IEEE Aerospace Conference Proceedings*, 2017. ISSN 1095323X. doi: 10.1109/AERO.2017.7943959.

Ali Bayramcavus, M. Cagri Kaya, and Ali H. Dogru. Interoperability of microservice-based systems. In *2021 13th International Conference on Electrical and Electronics Engineering (ELECO)*, pages 594–598, 2021. doi: 10.23919/ELECO54474.2021.9677712.

U. Narayan Bhat. *An Introduction to Queueing Theory*. Birkhäuser Boston, Boston, 2008. ISBN 978-0-8176-4724-7. doi: 10.1007/978-0-8176-4725-4. URL http://link.springer.com/10.1007/978-0-8176-4725-4.

Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review. *Empirical Software Engineering*, 26 (5), 2021. ISSN 15737616. doi: 10.1007/s10664-021-09999-9.

Amy Boyle. Effective Strategies for Kafka Topic Partitioning, 2021. URL https://newrelic.com/blog/best-practices/effective-strategies-kafka-topic-partitioning.

K Mani Chandy. *Event Driven Architecture*, pages 1040–1044. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_570. URL https://doi.org/10.1007/978-0-387-39940-9_570.

Liu Changfu and Liu Zhenyu. Research of Transaction Request Handling Queueing System in the e-Business Environment Based on Queuing Theory; Research of Transaction Request Handling Queueing System in the e-Business Environment Based on Queuing Theory. 2009. doi: 10.1109/APCIP.2009.280.

Junxiang Cheng and Ming Li. *Transient Queue Length Distribution of the Shortest Queue Model*. 2011. ISBN 9781612844596. doi: 10.1109/CECNET.2011.5768427.

Binildas Christudas. Distributed Messaging. In *Practical Microservices Architectural Patterns*, pages 105–145. Apress, 2019. doi: 10.1007/978-1-4842-4501-9_6.

Circuit Breaker. Akka Documentation Version 2.6.19. Circuit Breaker. URL https://doc.akka.io/docs/akka/current/common/circuitbreaker.html.

Derek DeJonghe. *Load Balancing in the Cloud*. 2018. URL https://learning.oreilly.com/library/view/load-balancing-in/9781492038009/.

Dunith Dhanushka. Understanding Kafka Topic Partitions, 2021. URL https://medium.com/event-driven-utopia/understanding-kafka-topic-partitions-ae40f80552e8.

Philippe Dobbelaere and Kyumars Sheykh Esmaili. Industry paper: Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations. *DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems*, (June):227–238, 2017. doi: 10.1145/3093742.3093908.

Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, 1986. ISSN 00985589. doi: 10.1109/TSE.1986.6312961.

Dirk Fahland. *Encyclopedia of Big Data Technologies*. Springer International Publishing, Cham, 2019. ISBN 978-3-319-77524-1. doi: 10.1007/978-3-319-77525-8. URL http://link.springer.com/10.1007/978-3-319-77525-8.

Tom Gregory. 3 surprising facts about AWS SQS FIFO queues, 2021. URL https://tomgregory.com/3-surprising-facts-about-aws-sqs-fifo-queues/.

Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pages 243–246, 2017. doi: 10.1109/ICSAW.2017.11.

Sören Henning and Wilhelm Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research*, 25:100209, 2021. ISSN 22145796. doi: 10.1016/j.bdr.2021.100209. URL https://doi.org/10.1016/j.bdr.2021.100209.

Nikolas Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. *International Conference on Autonomic Computing*, pages 23–27, 01 2013.

Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. *9th International Conference on Information and Communication Technology Convergence: ICT Convergence Powered by Smart Intelligence, ICTC 2018*, pages 257–259, 2018. doi: 10.1109/ICTC.2018.8539409.

Luke Hunter, Niveditha Rao, Eric Schow, and Thy Ton. How CrowdStrike Addressed Uneven Partition Lag in Kafka, 2021. URL https://www.crowdstrike.com/blog/how-crowdstrike-addressed-uneven-partition-lag-in-kafka/.

Kasun Indrasiri and Prabath Siriwardena. *Inter-Service Communication*, pages 63–88. Apress, Berkeley, CA, 2018. ISBN 978-1-4842-3858-5. doi: 10.1007/978-1-4842-3858-5_3. URL https://doi.org/10.1007/978-1-4842-3858-5_3.

James Lewis and Martin Fowler. Microservices, mar 2014. URL https://www.martinfowler.com/articles/microservices.html.

Thorben Janssen. Communication Between Microservices: How to Avoid Common Problems, 2017. URL https://stackify.com/communication-microservices-avoid-common-problems/.

Kafka documentation. Kafka 3.2 Documentation. URL https://kafka.apache.org/documentation.

Kafka protocol. Kafka protocol guide. URL https://kafka.apache.org/protocol.html.

Isıl Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdogan. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180, 2021. ISSN 01641212. doi: 10.1016/j.jss.2021.111014.

Hamzeh Khazaei, Jelena Misic, and Vojislav B. Misic. Performance analysis of cloud computing centers using M/G/m/m+r queuing systems. *IEEE Transactions on Parallel and Distributed Systems*, 23(5):936–943, 2012. ISSN 10459219. doi: 10.1109/TPDS.2011.199.

Pakorn Kookarinrat and Yaowadee Temtanapat. Design and implementation of a decentralized message bus for microservices. *2016 13th International Joint Conference on Computer Science and Software Engineering, JCSSE 2016*, 2016. doi: 10.1109/JCSSE.2016.7748869.

Jakub Korab. Message Distribution and Topic Partitioning in Kafka, 2015. URL https://www.jakubkorab.net/2015/12/message-distribution-and-topic-partitioning-in-kafka.html.

Seda Kul and Ahmet Sayar. A Survey of Publish/Subscribe Middleware Systems for Microservice Communication. *ISMSIT 2021 - 5th International Symposium on Multidisciplinary Studies and Innovative Technologies, Proceedings*, pages 781–785, 2021. doi: 10.1109/ISMSIT52890.2021.9604746.

Seda Kul, Isabek Tashiev, Ali Sentas, and Ahmet Sayar. Event-Based Microservices with Apache Kafka Streams: A Real-Time Vehicle Detection System Based on Type, Color, and Speed Attributes. *IEEE Access*, 9:83137–83148, 2021. ISSN 21693536. doi: 10.1109/ACCESS.2021.3085736.

Charles A Lapadula and Hanoch Levy. Customer delay in very large multi-queue single-server systems. *Performance Evaluation*, 26:201–203, 1996.

Volker Liermann and Claus Stegmann, editors. *The Digital Journey of Banking and Insurance, Volume III. Data Storage, Data Processing and Data Analysis*, volume III. 2021. ISBN 9783030788209.

Alan Megargel, Christopher M. Poskitt, and Venky Shankararaman. Microservices Orchestration vs. Choreography: A Decision Framework. *Proceedings - 2021 IEEE 25th International Enterprise Distributed Object Computing Conference, EDOC 2021*, pages 134–141, 2021. doi: 10.1109/EDOC52215.2021.00024.

Peter Membey, David Hows, and Eelco Plugge. *Practical Load Balancing: Ride the Performance Tiger*. 2012. URL https://learning.oreilly.com/library/view/practical-load-balancing/9781430236801/Titlepage.html.

Marko Milosavljević, Milica Matić, Neven Jović, and Marija Antić. Comparison of Message Queue Technologies for Highly Available Microservices in IoT. pages 6–9, 2021.

Nakadi Manual. Nakadi. URL https://nakadi.io/manual.html.

Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, and Sebastian Geiger. Evaluating and Improving Microservice Architecture Conformance to Architectural Design Decisions. pages 188–203. 2021. ISBN 9783030914301. doi: 10.1007/978-3-030-91431-8_12. URL https://link.springer.com/chapter/10.1007/978-3-030-91431-8_12https://link.springer.com/10.1007/978-3-030-91431-8_12.

Hugo Filipe Oliveira Rocha. *Dealing with Concurrency and Out-of-Order Messages*, pages 227–273. Apress, Berkeley, CA, 2022. ISBN 978-1-4842-7468-2. doi: 10.1007/978-1-4842-7468-2_6. URL https://doi.org/10.1007/978-1-4842-7468-2_6.

Vinicius Feitosa Pacheco. *Microservice Patterns and Best Practices*. 2018. ISBN 978-1-78847-403-0. URL https://learning.oreilly.com/library/view/microservice-patterns-and/9781788474030/.

Jonathan Prados-Garzon, Pablo Ameigeiras, Juan J Ramos-Munoz, Jorge Navarro-Ortiz, Pilar Andres-Maldonado, and Juan M Lopez-Soler. Performance Modeling of Softwarized Network Services Based on Queuing Theory With Experimental Validation; Performance Modeling of Softwarized Network Services Based on Queuing Theory With Experimental Validation. 2021. doi: 10.1109/TMC.2019.2962488. URL www.ieee.org/publications/rights/index.html.

Connected Retail. What is Connected Retail? URL https://www.connected-retail.com/how-it-works.

Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze. Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017*, pages 130–137, 2017. doi: 10.1109/QRS-C.2017.28.

Khaled Salah and Raouf Boutaba. Estimating service response time for elastic cloud applications. In *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 12–16, 2012. doi: 10.1109/CloudNet.2012.6483647.

Dharmendra Shadija, Mo Rezai, and Richard Hill. Microservices: Granularity vs. Performance. *UCC 2017 Companion - Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*, pages 215–220, 2017. doi: 10.1145/3147234.3148093.

Vindeep Singh and Sateesh K Peddoju. Container-based Microservice Architecture for Cloud Applications. In *International Conference on Computing, Communication and Automation (ICCCA2017)*, pages 847–852, 2017. ISBN 9781509064717. URL https://www.docker.com.

Francois Spies. Modeling of optimal load balancing strategy using queueing theory. *Microprocessing and microprogramming*, 41(8-9):555–570, 1996. ISSN 01656074. doi: 10.1016/0165-6074(95)00006-2.

Akshitha Sriraman and Thomas F Wenisch. µTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, oct 2018. USENIX Association. ISBN 978-1-939133-08-3. URL http://www.usenix.org/conference/osdi18/presentation/sriraman.

Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018-Janua (March):221–232, 2018. doi: 10.5220/0006798302210232.

Maxim Tschumak and Thomas Frauenstein. Developing Zalando APIs, 2019. URL https://engineering.zalando.com/posts/2019/04/developing-zalando-apis.html.

Huseyin Unlu, Samet Tenekeci, Ali Yıldız, and Onur Demirors. Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 244–251, 2021. ISBN 9781665427050. doi: 10.1109/SEAA53835.2021.00038. URL https://lee.iyte.edu.tr/en/graduate-application/application-to-.

J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. O. Ocharán-Hernández. Patterns Related to Microservice Architecture: a Multivocal Literature Review. *Programming and Computer Software*, 46(8):594–608, 2020. ISSN 16083261. doi: 10.1134/S0361768820080253.

Mark van der Boor, Sem Borst, and Johan van Leeuwaarden. Optimal hyper-scalable load balancing with a strict queue limit. *Performance Evaluation*, 149-150:102217, 2021. ISSN 01665316. doi: 10.1016/j.peva.2021.102217. URL https://doi.org/10.1016/j.peva.2021.102217.

Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, 2017. ISSN 18632394. doi: 10.1007/s11761-017-0208-y.

Cong Wan, Cuirong Wang, and Shuo Jia. Priority scheduling in mapreduce based on queuing theory. In *2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 428–431, 2013. doi: 10.1109/IMCCC.2013.98.

Fangzhou Wan, Xiaojun Wu, and Qin Zhang. Chain-Oriented Load Balancing in Microservice System. *2020 World Conference on Computing and Communication Technologies, WCCCT 2020*, pages 10–14, 2020. doi: 10.1109/WCCCT49810.2020.9169996.

Sean Whitesell, Rob Richardson, and Matthew D. Groves. Microservice Messaging. In *Pro Microservices in .NET 6*, pages 105–135. Apress, Berkeley, CA, 2022. doi: 10.1007/978-1-4842-7833-8_5. URL https://learning.oreilly.com/library/view/pro-microservices-in/9781484278338/html/480120_1_En_5_Chapter.xhtmlhttps://link.springer.com/10.1007/978-1-4842-7833-8_5.

Han Wu, Zhihao Shang, and Katinka Wolter. Performance prediction for the apache kafka messaging system. *Proceedings - 21st IEEE International Conference on High Performance Computing and Communications, 17th IEEE International Conference on Smart City and 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019*, pages 154–161, 2019. doi: 10.1109/HPCC/SmartCity/DSS.2019.00036.

Huimin Xiao and Guozheng Zhang. The queuing theory application in bank service optimization. In *2010 International Conference on Logistics Systems and Intelligent Management (ICLSIM)*, volume 2, pages 1097–1100, 2010. doi: 10.1109/ICLSIM.2010.5461127.

Eric Yuan. Architecture interoperability and repeatability with microservices: An industry perspective. *Proceedings - 2019 IEEE/ACM 2nd International Workshop on Establishing a Community-Wide Infrastructure for Architecture-Based Software Engineering, ECASE 2019*, pages 26–33, 2019. doi: 10.1109/ECASE.2019.00013.

Rong Zeng, Xiaofeng Hou, Lu Zhang, Chao Li, Wenli Zheng, and Minyi Guo. Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities. *Frontiers of Computer Science*, 16 (6):166106, 2022. ISSN 2095-2236. doi: 10.1007/s11704-020-0072-3. URL https://doi.org/10.1007/s11704-020-0072-3.