Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Mass Communication System: A State-of-Art Practice of Event Stream Processing

Author: Boyuan Xiao (VU: 2722126, UvA: 14733242)

1st supervisor:	Adam S.Z. Belloum	
daily supervisor:	Gustavo Guevara	(Virtuagym B.V.)
2nd reader:	Xin Zhou	

A thesis submitted in fulfillment of the requirements for *the joint UvA-VU Master of Science degree in Computer Science*

June 25, 2024

"I am the master of my fate, I am the captain of my soul"

from Invictus, *by William Ernest Henley*

Abstract

With the growth of its customer and code bases, Virtuagym's developers are encountering increasing challenges in managing and scaling their monolithic fitness software platform. To address these issues, the developers applied the Strangler Pattern to extract features from the original software platform into microservices. The Mass Communication System, one of these microservices, handles the delivery of time-sensitive messages such as push notifications and emails. It aims to deliver a massive volume of messages with minimal delay while providing real-time in-depth metrics, by leveraging an Event-Driven Architecture. This thesis presents the detailed design of the Mass Communication System as a state-of-the-art practice in Event Stream Processing, along with several experiments demonstrating its performance. The experimental results show that the Mass Communication System outperforms the legacy system in both latency and scalability tests.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Glossary

API	Application Programming Interface
APN	Apple Push Notification Service, notification delivery service for iOS devices
AWS	Amazon Web Services
CDC	Change Data Capture
EC2	Elastic Compute Cloud, a web service that provides resizable computing capability
EDA	Event-Driven Architecture
EKS	Elastic Kubernetes Service, an Amazon managed Kubernetes service
ETL	Extraction, Transformation and Loading
FaaS	Function-as-a-Service
FCM	Firebase Cloud Message, notification delivery service for Android devices
mass comm	the Mass Communication System
MSK	Managed Streaming for Apache Kafka, an Amazon managed Kafka service
REST	REpresentational State Transfer
SEP	Stream Event Processing
SNS	Simple Notification Service, notification delivery service offered by Amazon
vg-monolith	the monolithic fitness software platform created by Virtuagym

1 Introduction

The design of software systems has undergone a dramatic revolution in recent years. The trend of developing software systems in a distributed fashion, rather than in a monolithic way, continues to grow. For monolithic architectures, as the name suggests, all functionality is encapsulated into one single application, where its modules cannot be executed independently(1). Although a monolithic architecture offers many benefits, such as simplicity in development, ease of adopting radical changes, straightforward testing and deployment, and easy to scale, it also introduces challenges such as a prolonged development cycle and the risk of being locked into an increasingly obsolete technology stack. Additionally, the method of scaling up monolithic software often involves setting up more than one instances and masking them behind a load balancer. However, this approach is not resource-friendly when different software features have varying resource requirements. For example, the backend server of a food delivery application needs to store large restaurant data in memory, consuming significant DRAM, while its image processing module performs ideally with sufficient CPU power(2).

In contrast, software applications in distributed architectures are divided into sub-modules based on functionality. Each sub-module is typically designed to be independently deployable. As one of the most famous paradigm of building distributed architecture, microservice architecture thrives in the industry over last decades. Microservice architecture comprises multiple microservices, which are independently releasable services modeled around a business domain. A service encapsulates functionality and make it accessible to other services via networks(3). Many big organizations, including Netflix, Amazon and eBay, migrated their applications from monolithic architecture to microservices architecture for its various advantages, i.e. maintainability, reusability, scalability, availability and automated deployment(4).

As a technological company with almost 20 years of history, Virtuagym[1] found itself in a similar situation where the migration of its software application to a new architecture is necessary. Since 2008, Virtuagym invested huge amount of resources into its fitness software platform (refered as *vg-monolith* by its developers), which implies that this monolithic system is getting increasingly larger and potentially unmanageable in the future. Not only the functionality of the system is increasing, the number of customer is also rising in a fast pace. Challenges introduced by monolithic architecture are imminent. To tackle these problems, Virtuagym decided to use the Strangler Pattern(2) to divide the monolith system into microservices. The Strangler Pattern, as shown in Figure 1.1, suggests a way to migrate a legacy monolithic system to microservice architecture by incrementally developing a new (strangler) application around the legacy application. The strangler application, which adopts a microservice architecture, consists of services that implement either functionality previously resided in the monolith or new features. The Mass Communication System (*mass comm*) presented in this thesis is one of the microservices created by following the Strangler pattern.

The Mass Communication System (*mass comm*) extracts the functionality of sending and generating metrics for push notification and email from *vg-monolith*. Considering there are more than one million active daily users, *mass comm* should be able to handle significant amount of data. In addition, push notifications and emails must be delivered to users promptly to ensure timeliness. Therefore, high scalability and low latency are the two most important non-functional requirement of the system. To achieve this, the architecture team of Virtuagym decided to adopt Event-Driven Architecture (EDA) when designing *mass comm*. Similar to microservices, EDA is consist of several high-cohesive components that asynchronously react to events and perform a specific task(6). More and more people believe that EDA is a promising solution to the problem mentioned above. By nature, components of EDA are loosely coupled, which, in turn, enables high scalability. Low latency is also assured since systems reacts to each arriving event. In *mass comm*, requests for sending push notifications or emails are treated as events. Moreover, successfully delivered push notifications or emails are also seen as events that need to

[1]https://business.virtuagym.com/about-us/

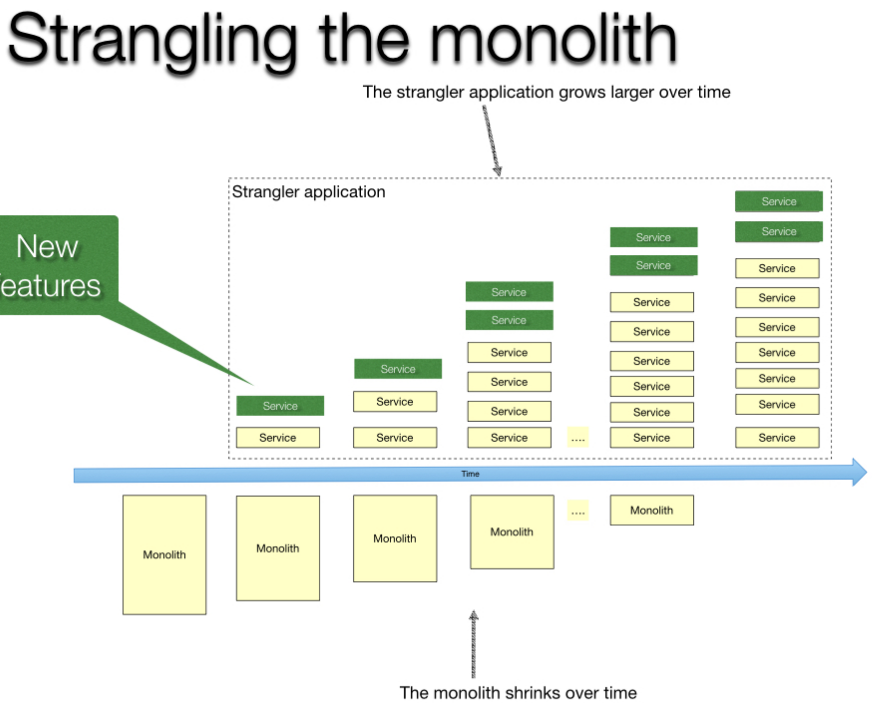


Figure 1.1: The strangler pattern(5). The monolithic system is being divided into microserverices over the time. New features are also implemented as independent services instead of parts of the monolithic system.

be processed to generate the metrics. The thesis presents *mass comm* as a state-of-the-art practice for constructing a highly scalable event stream processing system. This is achieved by leveraging industry-favored technologies, along with conducting experiments to demonstrate its capabilities. In addition, to show the main focus of this thesis, we list the following research questions:

- **Does *mass comm* maintain low latency as expected from an EDA?** The transition from a monolithic architecture to a distributed one would introduce extra over-the-network communication. As a result, the total process time needed to finish a request could also increase. Thus, we aim to uncover whether such transition could undermine the low latency characteristic.

- **Does *mass comm* possess high scalability, which is necessary to emit significant amount of messages, such as 100,000 and 1,000,000.** It is essential for *mass comm* to have high scalability, considering the amount of users Virtuagym has.

Although some of the design decisions (employing Kafka and ksqlDB) were already made before I joined the team, my contribution to *mass comm* includes designing and implementation of metrics feature as well as the design of experiments.

The structure of the thesis is as follow: Chapter 2 introduces the key concepts and technologies that play important roles in *mass comm.* While providing an overview of the terms, Chapter 2 also compares them to some alternatives and explains the reasoning behind choosing them. In Chapter 3, a comprehensive review on existing related literature is given. Chapter 4 presents the design and implementation process of *mass comm.* Experiments and corresponding results are displayed in Chapter 5. Then, we discuss about the outcomes and potential limitations of *mass comm* in Chapter 6. Finally, we give our conclusions in Chapter 7.

2

Background

In this chapter, an introduction to the underlying technologies of *mass comm* is given. For certain technologies, we also provide a detailed comparison with some alternatives, along with the reasons and benefits associated with choosing them. Section 2.1 introduces Event-Driven Architecture and its building-block technologies, which is foremost important as it defines the basic architecture of *mass comm.* Introduction of serverless computing, which is the approach we adopted to implement *mass comm*, is given in Section 2.2. Section 2.3 and Section 2.4 introduce Elasticsearch and Amazon SNS as key technologies for enhancing performance.

2.1 Event-Driven Architecture

The Event-Driven Architecture consist of highly decoupled, single-purpose event processing components that asynchronously receive and process events, which is an abstraction of a notable thing that happens inside or outside your business.(6)(7). The highly scalable and timely responsive nature of Event-Driven Architecture (EDA) plays an essential role in the design of *mass comm.* As a distributed asynchronous architecture pattern, EDA has evolved in recent decades and gained increasing attention from the industry(8). One of the prominent application scenario of EDA is Stream Event Processing (SEP) (7), which is acknowledged as a solution to fill the limitations created by Batch Processing. Batch Processing is an important building block in our quest to build reliable, scalable, and maintainable applications(9). As one of the most successful algorithm of Batch Processing, MapReduce was even called “the algorithm that makes Google so massively scalable”(10).

However, MapReduce is inherently designed for high throughput batch processing of big data that take several hours and even days, while recent demands are more centered on jobs and queries that should finish in seconds or at most, minute(11)(12). Another weak point of the Batch Processing paradigm is that the input data of Batch Processing systems are usually required to be bounded—i.e., of a known and finite size. Meanwhile in reality, a lot of data is unbounded because it arrives gradually over time(9). Therefore, in such cases, SEP excels with its advantage of low-latency response while maintaining high scalability.

Mark Richards(6) defines two topologies that are used in most EDA implementations. The Mediator Topology and Broker Topology. Figure 2.1 illustrates the general mediator topology of the event-driven architecture pattern. In a Mediator Topology, components are categorized as follows: Event Queue, Event Mediator, Event Channel and Event Processor. As a singleton instance, Event Mediator receives the initial event and orchestrates that event by sending additional asynchronous events to event queues. It is essential to note that the Event Mediator does not perform the business logic necessary to process the initial event. In contrast to the Mediator Topology, there is no centralized event mediators in the Broker Topology. Instead, events flow in a distributed manner across the Event Processor components. As shown in Figure 2.2, each Event Processor directly publishes and receives events from other Event Processors without going through a common component.

There are various pros and cons to consider when choosing between the two topologies for implementing an EDA. An EDA system implemented by using the Mediator Topology could suffer from single-point-failure and performance bottleneck, both introduced by the centralized Event Mediator. Meanwhile, systems that have adopted the Broker Topology must cope with the unavailability of each Event Processor. The high level design of *mass comm* resembles the Mediator Topology. To address the issues faced by the Mediator Topology, we employ Apache Kafka, a reliable and massively scalable message broker, as the Event Mediator.

2.1.1 Apache Kafka

Apache Kafka[1] is a distributed data streaming platform favored by thousands of companies including many well-known technological companies(13)(14). Thanks to its distributed nature, Kafka clusters have the capability of generating millisecond-level responses under massive workloads without worrying about scalability and availability. User applications,

[1]https://kafka.apache.org/

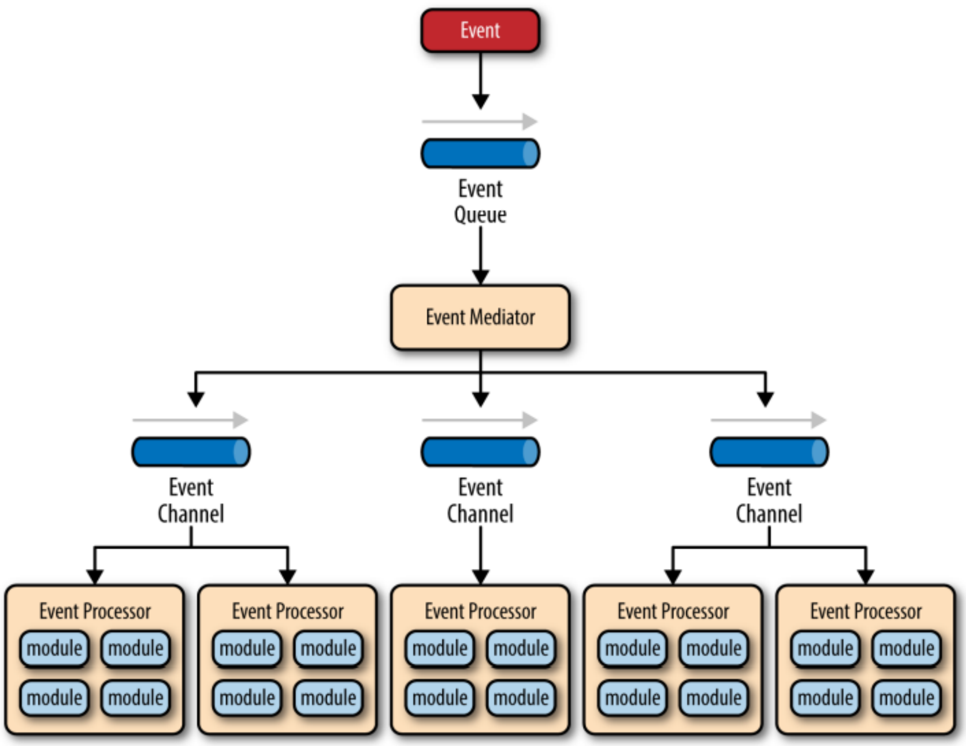


Figure 2.1: The mediator topology(6). A centralized event mediator distributes events to different event processors.

referred to as producers when sending events and consumers when receiving them, can establish a transparent message channel by leveraging the simple communication APIs provided by the Kafka client library. In addition, there are several powerful tools and plugins in the Kafka ecosystem that enables versatile applications, including Kafka Stream[1] and Kafka Connect[2].

Figure 2.3 gives an example of how producers and consumers interact with Kafka. Event data generated by a producer is published into different Kafka topics. Upon the arrival of the event, Kafka notifies the consumers that subscribed to the topic, providing them with the content of the event. With multiple producers and consumers communicating through a single Kafka cluster, high availability is still well preserved thanks to the distributedly deployed Kafka brokers. A Kafka cluster consists of one or more Kafka brokers, and each Kafka broker holds zero or more partitions of the same Kafka topic, depending on the replication factor. The number of partitions and replication factor can be designated when

[1]https://docs.confluent.io/platform/current/streams/overview.html
[2]https://docs.confluent.io/platform/current/connect/index.html

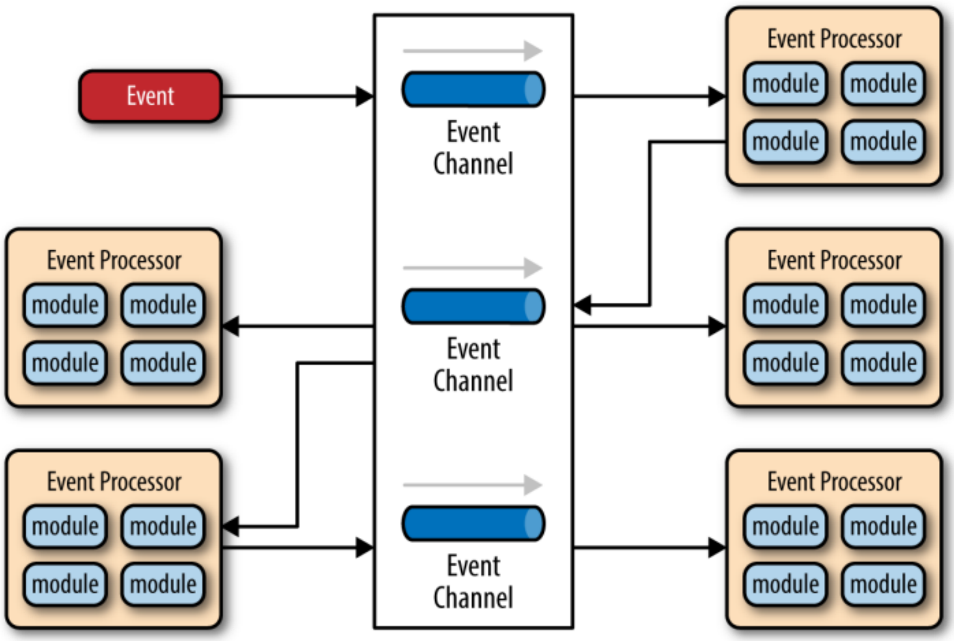


Figure 2.2: The broker topology(6). No centralized event distributor in this topology.

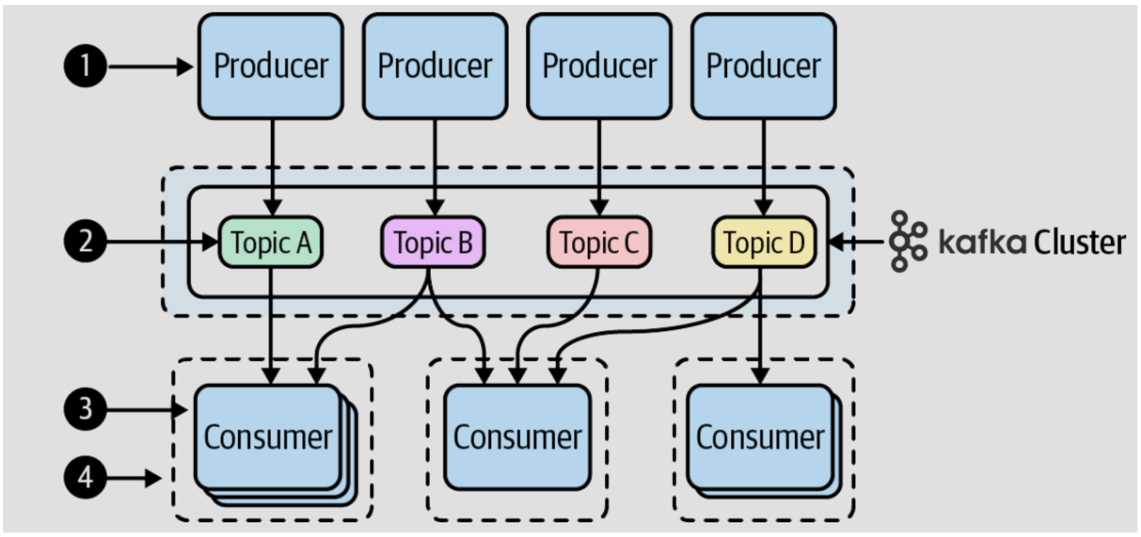


Figure 2.3: How Producers and Consumers Interact with Kafka(15)

creating the topic. In Figure 2.4, an example of a Kafka topic with 3 partitions and replication factor of 2 is given. Having more than one partitions for one topic eliminates concerns about insufficient storage for that topic. When a topic is initialized with a replication factor bigger than 1, each partition is replicated by that factor and each replica is stored in different brokers. Out of the 2 replicas, one replica acts as the lead replica for the other replica. In such a way, each partition of a event has n replicas and can afford n-1 failures to guarantee message delivery(16). Inside each partition, the events are stored using an append-only commit log structure, as shown in the Figure 2.5. Consumers locate desired events within the partition using offsets, which are maintained by consumers themselves.

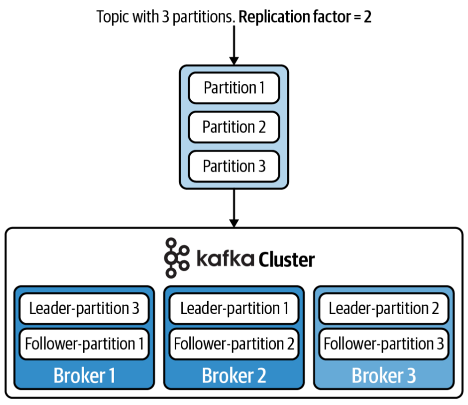


Figure 2.4: Each Topic Replication is Stored in Different Brokers(15)

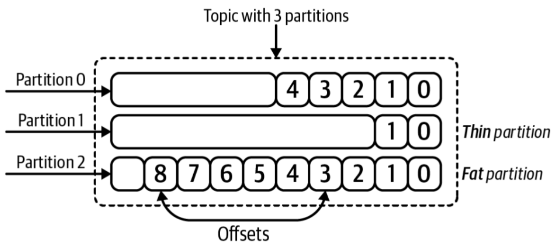


Figure 2.5: Events in Different Partitions are Consumed at Different Paces(15)

We also compared Kafka with other popular message brokers when making our design decisions. Works by Vineet et al.(17) and Aditya et al.(18) show comprehensive results of experiments for comparing Kafka and Rabbit MQ, which is a low latency message queue tha implements Advanced Message Queuing Protocol (AMQP). The results show that Kafka has higher throughput when the message payloads are short, which is ideal for *mass comm* considering the significant number of users and events flowing within *mass comm* are also small. In addition to that, Kafka also performs well in latency tests. Finally, instead of hosting our own Kafka cluster, we employ AWS MSK[1], which is a fully managed, highly available Apache Kafka service offered by Amazon Web Services (AWS). Utilizing MSK helps with setting up and managing the cluster without extra effort, and it can be seamlessly integrated into the Virtuagym ecosystem, which comprises services based on other AWS services.

2.1.2 Avro Schema and Schema Registry

Topics in Kafka are similar to tables in databases; both are the smallest units used for grouping related data. However, the most significant difference between topics and tables

[1]https://aws.amazon.com/msk/

lies in whether they have a predefined schema for data. While data in the same table all has the same format, Kafka does not impose any particular schema for the data in the same topic. Instead, Kafka stores the raw bytes of data(15). Although the design of not imposing a schema offers flexibility, the risk of maintenance issues increases when there are multiple pairs of producers and consumers. Whenever a producer intends to introduce changes to the event format, all corresponding consumers should also adapt to those changes. Achieving this would require much more effort without imposing a data schema. We also considered this issue when designing *mass comm*. Finally, we decided to utilize Avro schema along with a schema registry.

Apache Avro[1] is a compact binary data serialization format(19). According to the official documentation of Apache Avro(20), the Avro data format supports a variety of data structures including array, map and nested complex types. Defining an Avro schema is also straightforward since Avro's syntax is the same as JSON. Therefore, creating an Avro schema is equivalent to creating a JSON file filled with field names and associated data types. Not only within *mass comm*, but also almost all the events that go through Kafka and are produced by services owned by Virtuagym are serialized using Avro. As a result, we use Schema Registry as a solution to potential management problems. Schema Registry is a centralized repository for managing and validating schemas(21), which supports multiple popular schema formats including Avro, Protobuf[2] and JSON. With the assistance of Schema Registry, the versioning feature of Avro becomes even more formidable. Figure 2.6 gives a clear demonstration of how Schema Registry works.

Extensive deliberation preceded our decision to adopt Avro during the design phase of *mass comm*. We were ultimately attracted by the following benefits of using Avro:

- **Ease to use.** It is fairly simple to employ Avro and this is not only because its JSON syntax, which is familiar to most of the developers. Avro is a dynamically-typed schema, ensuring that the schema used during writing is always available when reading Avro data. This also eliminates the need of code generation, which facilitates the development process. In contrast, Protobuf schemas require code generation every time when schemas are modified(20).

- **High performance.** Since the schema is always present when data is being deserialized, the extra information used to encode data is significantly reduced. Thus,

[1]https://avro.apache.org/
[2]https://protobuf.dev/

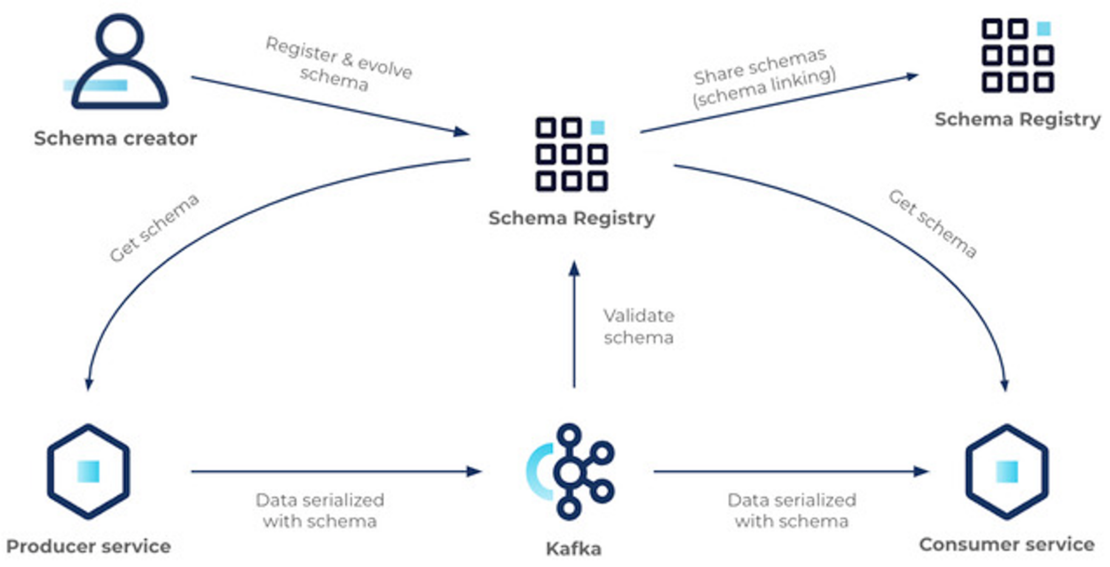


Figure 2.6: How Schema Registry Works(21)

the serialized data has a smaller size compared to Protobuf. This is also proved by the works done by Kazuaki Maeda(22). Kazuaki conducted several experiments to compare the performance of more than 10 popular object serialization libraries. The size of serialized data and the execution time for serializing and deserializing are used as metrics for the experiments. In the results, Avro excels in the size test without sacrificing too much speed.

2.1.3 Change Data Capture (CDC)

Change Data Capture (CDC) is a solution to achieve data synchronization between *vg-monolith* and *mass comm*. Data synchronization stands as a substantial problem for data systems that comprise a number of sub-systems. Typical examples in real-world scenarios include: an Online Transaction Processing (OLTP) database to handle user requests, a cache server to expedite common requests, a full-text index for search queries and a data warehouse for conducting business analytics. It is common that the timely update of data is considered a fundamental requirement of these system. Except for CDC, other solutions for data synchronization include dual write and the "Extraction, Transformation and Loading", which is usually referred as ETL process. While adopting dual write may require addressing issues caused by race conditions and paying extra attention to enhance fault-tolerance, the ETL process is recognized as a batch processing method, implying its incapability for low-latency processing.(9) However, upon considering CDC, we found studies that accelerate the ETL process by leveraging CDC technologies(23)(24). Big technology companies like LinkedIn have also invested efforts in implementing a low-latency CDC platform(25). Thus, we see CDC as a promising solution and shifted our main focus to exploring a reliable CDC solution when solving the data synchronization problem for *mass comm*.

The idea behind CDC is to identify the change of data from the source data store, which is usually a relational database, and propagate this change to the targeting data store across a network. There are several methods that can be employed to implement a CDC, which include timestamp-based method, trigger-based method and log based method(26)(27). The pros and cons of utilizing each method can be seen below:

- **Timastamp-based:** Using a timestamp to detect record changes is simple to use and implement. However, this could not detect records that are deleted. It also introduce computational overhead since CDC has to determine whether the record should be broadcast by comparing those timestamps.

- **Trigger-based:** A trigger refers to database triggers that invoke specific actions when certain conditions are met.(28). Although all kinds of data updates (INSERT, UPDATE and DELETE) can be captured by this method, the computational overhead still exists. In addition, using trigger requires schema changes.

- **Log-based:** The log-based approach leverages the change logs maintained by the source database. The biggest benefit of log-based approach is that no overhead is introduced while all types of changes can be captured. Unfortunately, it is not a perfect solution, as it requires efforts to decode the change logs produced by databases maintained by different vendors, which are often opaque black boxes. Moreover, target systems have to identify and eliminate any changes that were written to source databases but then rolled back.

After the above analysis of CDC implementations, we decided to adopt a existing CDC library that supports the log-based approach. Finally we choose to use Debezium[1]. Debezium is an open source distributed CDC platform that provides seamless integration with Kafka. Deploying a Debezium CDC connector on Kafka only requires adding a Debezium

[1]https://debezium.io/

plugin to a Kafka Connect instance and making a REST call to provide necessary configuration. Then, the CDC connector will stream the database changes that it captures to a specific Kafka topic. The list of supported data stores listed in Debezium's official documentation includes the majority of popular databases. The experiments conducted by JIŘÍ(29) also demonstrates the high throughput of Debezium.

2.1.4 ksqlDB

So far, introductions and rationales of the building blocks for the push notification and email delivery features of *mass comm* have been provided. To generate real-time metrics for the push notification and email within *mass comm*, a mature stream processing framework is required. In conclusion, we finally decided to use KsqlDB[1] as our stream processing framework.

ksqlDB is an open source event streaming database that was released by Confluent[2]. With Kafka Stream and Kafka Connect being its backbone, ksqlDB provides powerful stream processing features, including aggregation, group-and-aggregate, join and sort, which are essential in data stream processing(30). Additionally, it supports effortless integration to Kafka(15). Moreover, interacting with ksqlDB requires minimal proficiency in SQL syntax, which significantly enhances simplicity. ksqlDB establishes two abstractions to model data: stream and table. While a stream is a collection of historical sequences of events, a table maintains a snapshot of the latest value identified by the key in the event(31). After creating necessary streams or tables, user applications can issue *push queries* or *pull queries* against those entities. *Pull queries* operate similarly to key-based lookups commonly used in traditional relational databases. In contrast, *push queries* continuously send the latest updates to the subscriber until the query is terminated by the subscriber. Users can also stream the results of queries into a desired data store by leveraging ksql connectors.

Next, we outline the major benefits of ksqlDB along with the comparison with some other popular stream processing frameworks.

- **Simplicity.** The simplicity of ksqlDB primarily stems from its SQL-like interface. It not only drastically reduces the learning cost, but also simplifies source code management. Besides that, deploying a ksqlDB server is relatively straightforward as it can

[1]https://ksqldb.io/overview.html
[2]https://www.confluent.io/

be deployed in a standalone server fashion. On the other hand, using Apache Flink[1] in this project is may be an overkill. Employing Apache Flink introduces significant complexity due to its specialization in handling exceptionally large amounts of state and scale, often requires complex aggregations(15)(32).

- **Easy integration with Kafka.** Considering that Kafka serves as the center message ingestion point of *mass comm*, the selected framework should be able to integrated with Kafka without too much effort. ksqlDB satisfy the requirement as it is built on top of two essential components of Kafka: Kafka Stream and Kafka Connect. In contrast, Apache Spark Structured Streaming[2] is strongly tied to Apache Spark[3]. While Structured Streaming provides notable performance improvements over Flink and Kafka Streams, with strong scalability and fault tolerance(33), incorporating an entire Spark cluster may complicate the architecture, contradicting the goal of maintaining simplicity for ease of maintenance.

2.2 Serverless Computing

When implementing the actual business logic in *mass comm*, we followed serverless computing paradigm. The severless computing platform we chose is Amazon Lambda[4]. Serverless computing, also known as Function-as-a-Service (FaaS), simplifies application hosting for software developers by eliminating the need of deploying and scaling underlying infrastructure. The term 'serverless' does not imply the absence of an actual server. Instead, it refers to the abstraction of server management, where the control of the server running the 'function' is completely hidden by cloud providers. Modern serverless computing services provided by cloud providers offer better autoscaling, strong isolation, platform flexibility, and service ecosystem support. The pay-as-you-go pricing model is beneficial for applications with variable traffic patterns(34)(35).

Moreover, serverless computing fits perfectly in an EDA. When viewed from a broader perspective, serverless computing inherently follows an event-driven paradigm. Invoking a function with specific parameters can be identified as the arrival of an event within the system(36). Furthermore, events tend to be largely independent and stateless in nature, which makes them ideal candidates for event-driven and FaaS architectures(34).

[1]https://flink.apache.org/
[2]https://spark.apache.org/streaming/
[3]https://spark.apache.org/
[4]https://aws.amazon.com/lambda/

2.3 Elasticsearch

Elasticsearch[1] is a distributed, scalable, real-time search and analytics engine built on top of Apache Lucene[2]. Elasticsearch offers features such as high-performance full-text search and real-time analytics of structured data(37). The user base of Elasticsearch is also ginormous. According to March, 2024 figures of DB-Engines Ranking of Search Engines, Elasticsearch is the most popular search engine software(38). Elasticsearch plays an important role in *mass comm* as it facilitates the notification settings check. More details on this will be discussed in Chapter 4.

Eliminating the performance bottleneck is the biggest reason why we included Elasticsearch in the design of *mass comm*. If not properly addressed, searching in a tremendous user dataset could significantly impact overall performance, leading to a degraded user experience. The studies done by Doina et al.(39) and Mustafa et al.(40) demonstrate the remarkable searching speed of Elasticsearch. The former conducted comprehensive experiments to compare the performance of MySQL Document Store[3] and Elasticsearch. Elasticsearch not only presents a shorter query time in most of the speed tests but also consumes much smaller disk space. MySQL Document Store can only have a better performance when there are specially defined indices for columns, which are not necessary for Elasticsearch. Experiments in (40), which compare Apache Solr[4] and Elasticsearch, show similar results. However, Mustafa et al.'s conclusion does not indicate superiority of one approach over the other. Elasticsearch and Solr both have significant search speed. Gaps only emerge when the length of document data varies. It took Elasticsearch 30 minutes and Solr 43 minutes to index data consists of 40 million pieces with 5 to 20 words. When the size of input data was increased to 200 to 1000 words, Elasticsearch used 179 minutes while Solr used 119 minutes.

During the exploration of high performance search engines, we noticed that AWS offers a fully-managed OpenSearch service[5], which is based on OpenSearch[6], an open source, distributed search and analytics suite derived from Elasticsearch. By leveraging AWS OpenSearch, the deployment process is significantly simplified.

[1]https://www.elastic.co/
[2]https://lucene.apache.org/
[3]https://www.mysql.com/products/enterprise/document_store.html
[4]https://solr.apache.org/
[5]https://aws.amazon.com/opensearch-service/
[6]https://opensearch.org/

2.4 Amazon SNS

As the final building block of *mass comm*, Amazon Simple Notification Service[1] (SNS) takes on the responsibility of delivering push notification and email to end users. Amazon SNS is a fully managed publish/subscribe service designed for application-to-application (A2A) and application-to-person (A2P) messaging. The A2P notification messaging offered by SNS comprises a wide range of destination platforms, including Android, iOS, Fire, Windows and Baidu devices. In addition, SNS is also capable of delivering emails and SMS texts. Compared to the implementation in *vg-monolith*, which involved using three different services for message delivery (Firebase Cloud Message for Android notifications, Apple Push Notification service for iOS notifications, and an SMTP server for email), utilizing SNS significantly reduces management costs.

[1]https://aws.amazon.com/sns/

3

Related Work

In this chapter, some similar studies are discussed. The similarity may lie in the goal of the project and the technologies used. Section 3.1 and 3.2 show related works that significantly influenced the design of *mass comm*. More works that leverages both Kafka and ksqlDB are given in Section 3.3 to demonstrate the uniqueness of *mass comm*. For each academic work, we give a brief introduction and discuss how these works influenced our design of *mass comm* or how they differ from our work.

3.1 Works on Real-Time Stream Processing

Long before the arrival of Kafka, the Extract, transform, and load (ETL) based batch processing jobs were the utmost primary data processing technique employed at LinkedIn(41). The continuous ETL pipeline generates valuable insights based on user's activity data and drives several downstream features or applications that are essential for the ongoing operations of the business. One of these downstream application is generating operational system metrics. Quoting from the paper: 'For a large-scale consumer website, operational system metrics about server and service performance are business metrics.' However, the lengthy execution time of batch processing jobs, which is usually several hours, poses a significant challenge to collect real-time operational system metrics. As a result, a real-time activity data processing solution that can be seamlessly integrated into other existing systems is needed. The authors made first attempt by adopting ActiveMQ as a real-time message queue to feed the central logging system. Although it worked pretty well in some of the real-time performance tests, serious performance degradation appears when the system has to confront real production workload. The root cause of this problem is the amount of data is exceeding the memory capacity, hence random I/O comes into play. Therefore, they have to seek for another reliable solution. It was under such a context, Apache Kafka was born.

The extensive usage of Kafka at LinkedIn proves the importance of Kafka. The average message writes per day handled by Kafka is more than 10 billion. During peek traffic hour, the average workload goes up to 172,000 messages per second. In total, they see a ratio of roughly 5.5 messages consumed for each message produced, which means 55 billion messages are delivered to real-time consumers. These figures brought us huge confidence in Kafka. This gives us another reason to choose Kafka as our stream processing backbone.

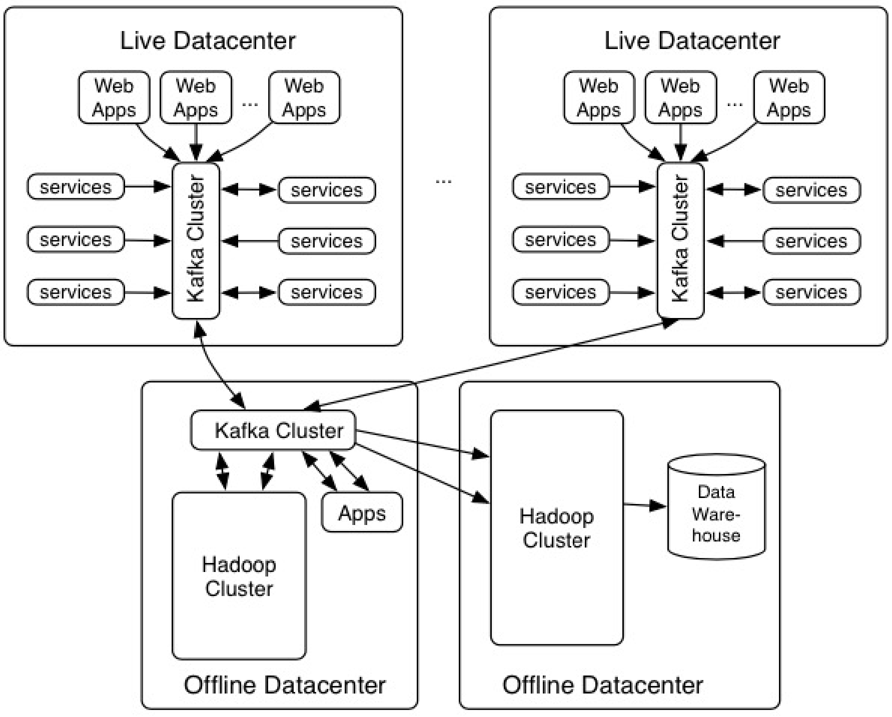


Figure 3.1: How Kafka Fits in LinkedIn's Architecture(41)

There are three important techniques adopted by the team to engineer high throughput for Kafka, including batch sending, shrinking data size and reliance on page cache. The technique of shrinking data size caught our attention when designing *mass comm* since this can also be easily employed. Designers of Kafka deems shrinking data as the most important performance technique as the bandwidth between data center facilities is acknowledged as one of the major bottlenecks. This bandwidth is more expensive per-byte to scale than disk I/O, CPU, or network bandwidth capacity within a facility. To tackle this bottleneck issue, they replaced initially employed XML format with the Apache Avro, which leads to a roughly 7 times smaller average data size. Moreover, a GZIP compression mechanism is also being utilized to further increase the throughput. Although the GZIP compression introduces higher CPU usage, the overall throughput is boosted by 30%. As a result, Avro also becomes the standardized message format of *mass comm* and we pay extra attention to the compression settings when configuring our MSK cluster.

3.2 Works on SQL Stream Processing

The study(42) presented by the researchers at IBM makes another attempt to leverage SQL syntax for real-time stream processing. The project serves as an extension of IBM's Cloud Data Engine[1], which was a SQL service only supports batch-oriented queries based on Apache Spark. With the contribution from the paper, Cloud Data Engine is now capable of ingesting and querying stream data provided by Apache Kafka. There is a fundamental difference between their work and ksqlDB. ksqlDB requires the data from a static data source to be transformed into a stream before they can be processed, i.e. using a read file utility to emit each line of a CSV file as a stream of events into a Kafka topic. Meanwhile, since Data Engine already has the capability of making SQL queries on batch data source, the requirement imposed by ksqlDB is no longer meaningful.

Figure 3.2 shows how SQL queries on stream data are done in Data Engine. While the rest of the figure remains clear, the part of Schema Registry, Metastore and Registry Proxy is intriguing. Metastore is a metadata storage that already exists in previous versions of Data Engine. The metadata includes table definitions containing schema information, the storage location of each table, the storage format and how the data is partitioned. Whenever SQL Query Execution receives a query from Service API Job Handler, it will request the table's schema definition from the Metastore to perform syntax checks. Since in the work presented by IBM's researchers, Data Engine includes functionalities related to stream data, the Metastore also contains schema of stream data now. As mentioned in Section 2.1.1, Kafka topics are schema-agnostic, which is why a Schema Registry has to be employed to reduce the management cost introduced by schema changes. To ensure Schema Registry fits in the overall design while making Metastore backward compatible,

[1]https://cloud.ibm.com/docs/sql-query?topic=sql-query-overview&locale=en

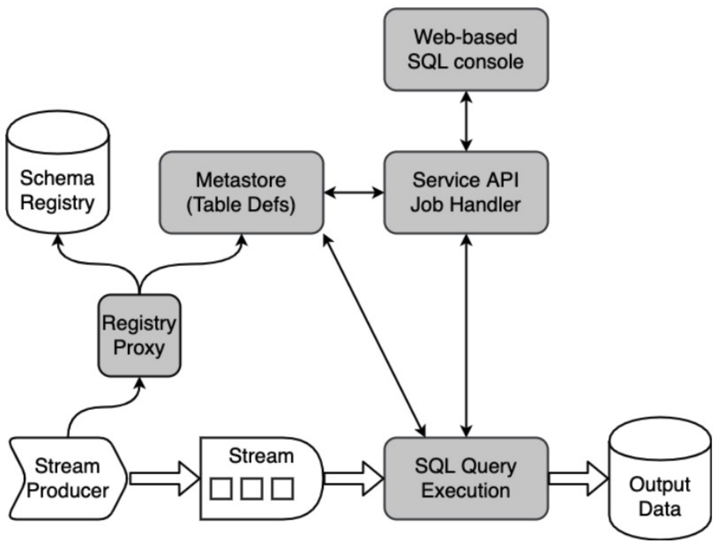


Figure 3.2: The Stream Query Happened in Data Engine(41)

IBM's researchers included Registry Proxy as a method to synchronize the schema changes between Schema Registry and Metastore. Registry Proxy guarantees the consistency by intercepting the schema creation or update request from Stream Producer and broadcast it to both Schema Registry and Metastore.

IBM's practice shows us how schema changes can be addressed properly in a distributed big data system. Although their final solution, which is incorporating three components to manage schemas from various sources, can be simplified by either extending the functionalities of Metastore to maintain schemas of stream data or migrating the old metadata to the Schema Registry, the idea heavily influenced our design of *mass comm*.

3.3 Studies Based on Kafka and ksqlDB

After examining the two most influential works, we present reviews of several similar studies that also leverage both Apache Kafka and ksqlDB for stream processing in this section.

The work by Adeyinka Akanbi(43) presents EStemd: which is a distributed stream processing framework for environmental monitoring. The framework ingests heterogeneous environmental data from different sources, including Internet of Thing (IoT) sensors, automated weather stations and legacy weather stations. The data flows through Kafka and is processed by KSQL, which is the precursor of ksqlDB, to support valuable decision-making. The biggest distinction between this work and the work by Adeyinka Akanbi is the imposition of event schemas and inclusion of a Schema Registry. Although imposing schemas would increase development cost under the context of using heterogeneous data sources, it can, however, reduce management costs and ensure transparency in the long term.

Sudeshna et al.(44) describe a real-time data management framework specifically designed to manage the information generated by smart meters in low-voltage networks. Data collected from voltage meters, which is processed and transformed by Kafka and ksqlDB, can be intended for a wide range of usage in power systems, such as Supervisory Control and Data Acquisition (SCADA) system and Advanced Metering Infrastructure (AMI). While a Schema-Registry-like component is also missing, the framework employs JSON format for event messages, which would potentially increase the performance overhead.

In the work by Adrian et al.(45) a solution that aims to provide real-time network traffic logging and analysis is given. The framework can either read in network packets from a static file or capture network packets from a live network interface using its own abstraction wrapped around the packet capture API provided by the Libtins library. An extensible design is also adopted in their framework where users can freely choose the type of processing they wish to apply on the captured packets and the means in which they wish to forward the data, thanks to the on-the-fly query functionality offered by ksqlDB.

In the end, there is another fundamental difference that distinguishes *mass comm* from the work mentioned above. The fact that *mass comm* is an extraction of functionality from a legacy system, *vg-monolith*. Thus, it enables us to compare the performance difference when switched to an EDA implementation. More details about this comparison can be found in later chapters about experiments.

4

Design and Implementation

In this chapter, we elaborate on the design and implementation detail of *mass comm*. In addition, the ksql queries that power the real-time analysis are also discussed.

4.1 Previous Architecture

Before deep dive into the design of *mass comm*, it is necessary to first look at the previous design, *vg-monolith*. As the name suggests, *vg-monolith* is a monolithic software platform comprising multiple functionalities, including the push notification and email delivery feature of *mass comm*. Although *vg-monolith* is deployed as a single process, steps taken to send a push notification or email involves asynchronous communications.

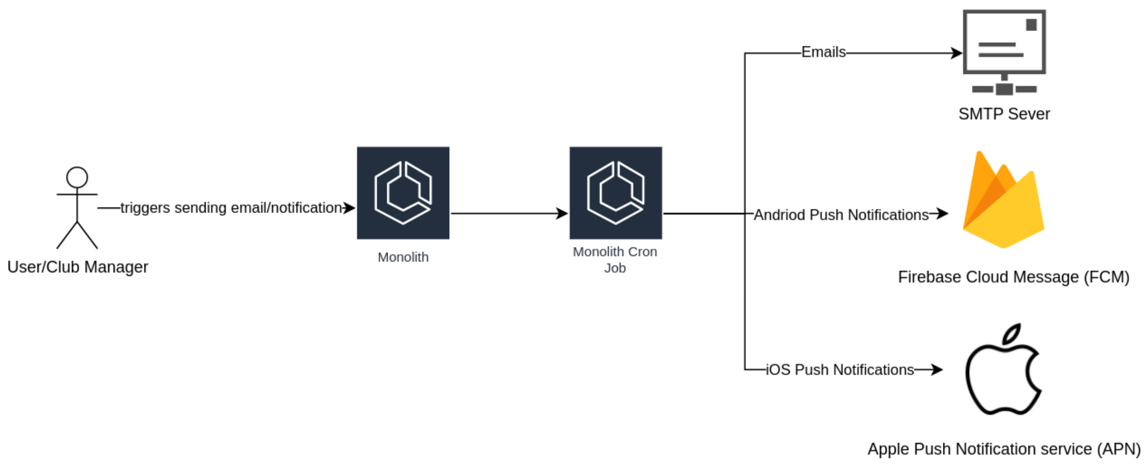

User/Club Manager
triggers sending email/notification
Monolith
Monolith Cron Job
Emails
SMTP Sever
Andriod Push Notifications
Firebase Cloud Message (FCM)
iOS Push Notifications
Apple Push Notification service (APN)

Figure 4.1: The Architecture of *The Vg-Monolith*

As shown in Figure 4.1, when a certain user request that could trigger the sending of a push notification or email arrives, *vg-monolith* will first check the user settings to determine if the user refuses to receive the message and build the message body. Next, a dedicated cron job will be spawned to asynchronously deliver the message to either Firebase Cloud Message (FCM), Apple Push Notification service (APN) or a Simple Mail Transfer Protocol (SMTP) server depending on the type of the message and targeting device.

4.2 Architecture Overview

The architecture of *mass comm* is shown in Figure 4.2. The business logic in *mass comm* consists of five Amazon Lambda functions, namely, Preference Checker, Builder, Delivery, Metrics Handler and Send Handler. In the new architecture, there are two sources of messages: *vg-monolith* and Send Handler. When a user request arrives at *vg-monolith*, a Kafka event will be emitted into a topic in the MSK cluster to trigger the following action in *mass comm*. On the other hand, Send Handler receives send requests of push notification or email either directly from users through a REST call or the Vue Web Frontend. Then, a Kafka event following the same schema as the events from *vg-monolith*, is published into another topic in the MSK. Preference Checker subscribes the topic to perform notification preference checks based on user's notification settings. If the check passes, Preference Checker sends a build event into a different topic, which is subscribed by Builder function. Similarly, a delivery event is published to the topic that Delivery listens to, carrying the notification message formatted with user's information by Builder function. Finally, upon the reception of the delivery event, Delivery function initiate REST calls, which are attached with user's device information, to Amazon SNS to make the actual delivery. The whole process of sending a push notification as a sequence diagram is given in Figure 4.3. The data used by each lambda function (user settings, user information and user device information) is fetched from Amazon OpenSearch, which is populated by Debezium CDC from the database of *vg-monolith*.

The metrics of push notification and email are generated by modeling the two topics as streams and make further ksql queries in ksqlDB. Output data of these queries is streamed into a Postgres database for persistence through a ksqlDB connector. The requests for metrics data are handled by Metrics Handler. Similar to Send Handler, these requests can be made directly through REST call to the handler or from the Vue Web Frontend.

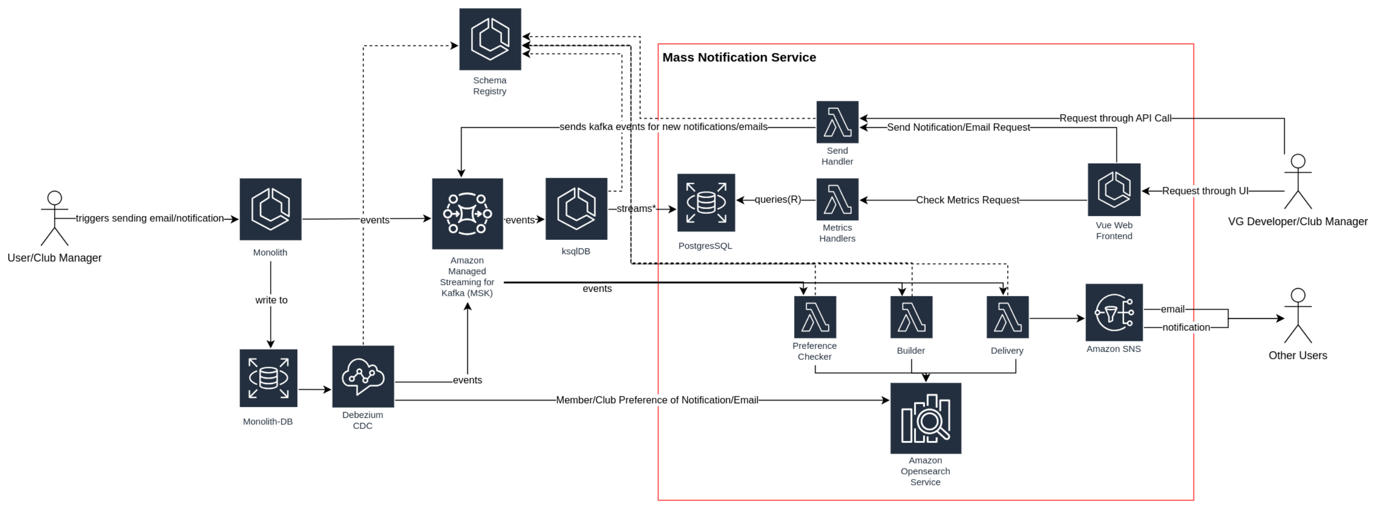


Figure 4.2: The Architecture of *Mass Comm*

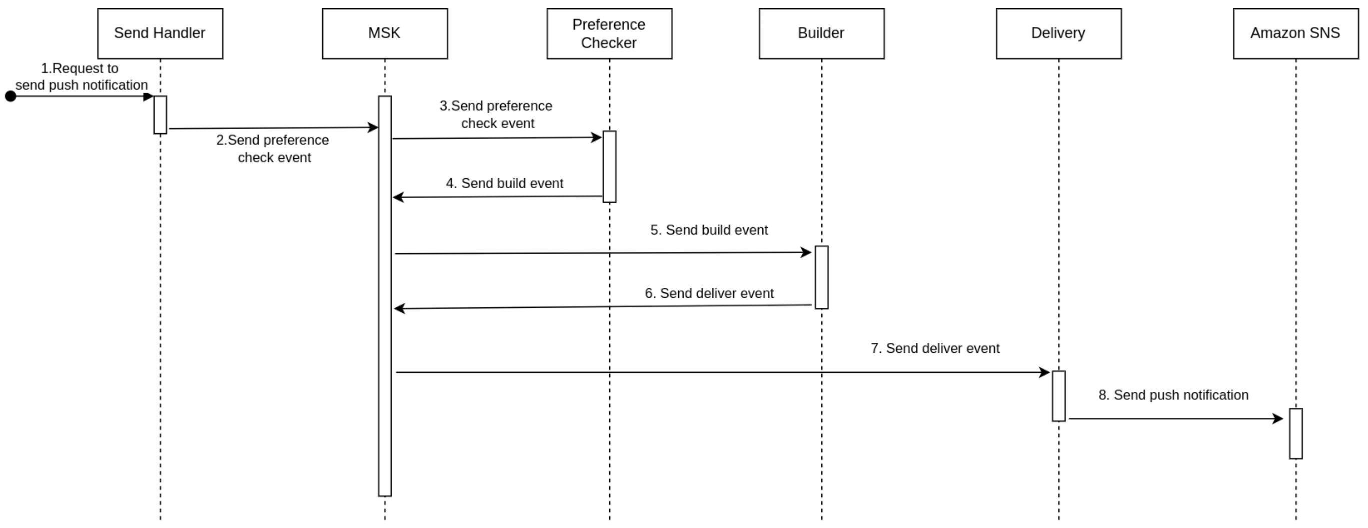


Figure 4.3: The Sequence Diagram of *Mass Comm*

The doted lines in Figure 4.2 denotes the serialization and deserialization happened in *mass comm*. Components that need to publish to or read from MSK will have to first make REST calls to Schema Registry to retrieve the latest schema. This also applies to ksqlDB when a special format is used in ksql queries. ksqlDB uses Kafka topics as intermediate store for each ksql query. The schema format, which is Avro in *mass comm*, is enforced when ksqlDB streams results of queries to Kafka topics.

4.3 ksqlDB Data Structure Design

mass comm collects various metrics about push notification and email to support business decision-making in Virtuagym. In this section, the ksql queries used to generate metrics of total push notifications sent per different time unit in a time period are discussed.

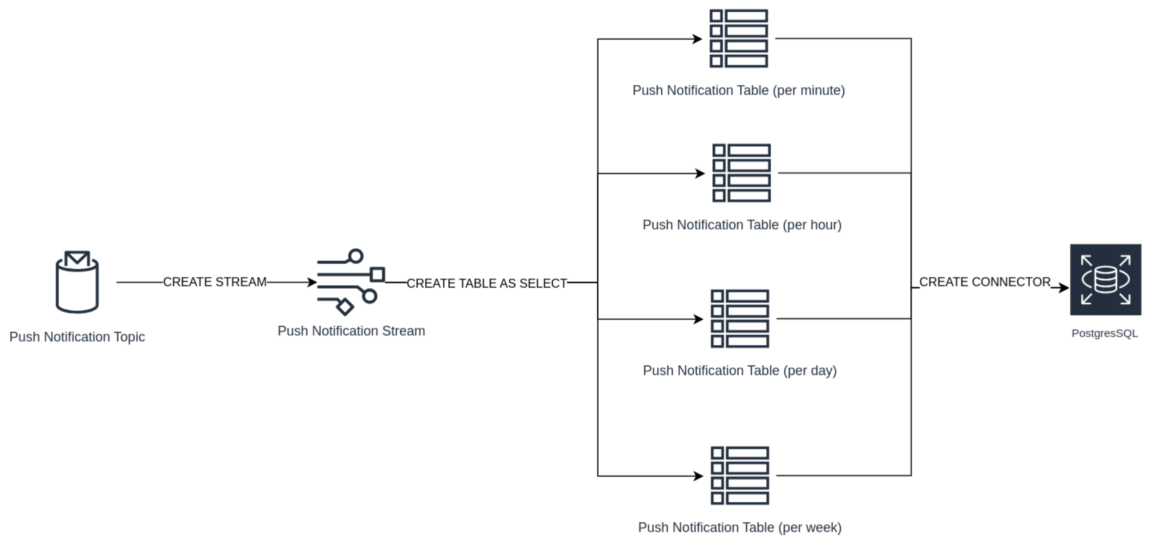

Push Notification Topic
CREATE STREAM
Push Notification Stream
CREATE TABLE AS SELECT
Push Notification Table (per minute)
Push Notification Table (per hour)
Push Notification Table (per day)
Push Notification Table (per week)
CREATE CONNECTOR
PostgresSQL

Figure 4.4: The ksqlDB Data Flow

The pipeline is shown in Figure 4.4. A ksql stream is first initialized in ksqlDB based on the Kafka topic containing the events of sending a push notification. Then, a ksql table with data grouped by the timestamp can be derived. Finally, the data is streamed into a Postgres database using a ksql connector. For example, to generate metrics of push notifications sent per minute, only three queries are needed.

```
# This creates a stream of send push notification events based on a kafka topic
CREATE STREAM pushnotification_stream
    WITH (KAFKA_TOPIC='pushnotification_send', VALUE_FORMAT='AVRO');

# This creates a table based on a select query from the stream
CREATE TABLE IF NOT EXISTS pushnotification_table_minute
    WITH (KAFKA_TOPIC='pushnotification_table_minute', FORMAT='AVRO')
AS SELECT
    type,
    club_id,
    timestamp - timestamp % 60000 AS timestamp,
    SUM(ARRAY_LENGTH(to_users)) AS count_total
    FROM pushnotification_stream GROUP BY type, club_id, timestamp - timestamp % 6000
    EMIT CHANGES;

# This creates a connector that streams the data in the table into a database
CREATE SINK CONNECTOR IF NOT EXISTS pushnotification_connector WITH(
    "connector.class"='io.confluent.connect.jdbc.JdbcSinkConnector',
    "connection.url"='jdbc:postgresql://host:port/db?user=user&password=password',
    "auto.create"='true',
    "topics"='pushnotification_table_minute',
    "key.converter"='io.confluent.connect.avro.AvroConverter',
    "value.converter"='io.confluent.connect.avro.AvroConverter',
    "key.converter.schema.registry.url"='https://schema-registry:8088',
    "value.converter.schema.registry.url"='https://schema-registry:8088',
    "insert.mode"='upsert',
    "pk.mode"='record_key',
    "pk.fields"='club_id,type,timestamp');
```

The stream created by first query is necessary to model the topic events to enable following processing. When doing a `SELECT` from that stream, the result will be excatly the same as original events in the topic. The second query, which creates a table, groups the data by the data by three fields, `club_id, type` and `timestamp` and sums the total number of push notifications based on the `to_users` field. The fields used for grouping will be the event key of the resulting record. Note that the timestamp is a UNIX timestamp in milliseconds and is rounded up to the nearest starting millisecond of the minute window. By changing how much the timestamp is rounded up, metrics based on different time unit can be generated as shown in Figure 4.4. Since events in the stream arrive continuously, result records in the table also need to be updated accordingly. Counterintuitively, this is done by publishing a new event with the new value, which is the value of `count_total`, into `pushnotification_table_minute` topic without deleting records with stale value, instead of modifying the same record in the topic like traditional databases. However, intermediate values are not shown when running a `SELECT` from that table. The result set will only contain the records with distinct keys and latest values, giving users the illusion of querying against a table in common databases. Finally, the third query creates the connector who makes the processed persistent in a database. The event source of the connector is the topic `pushnotification_table_minute` and the destination database is defined by `connection.url`. To ensure the intermediate values are excluded, the `insert.mode=upsert` must be present in the query. By doing so, the connector will insert a new record if the event key does not yet exist in the database table, and update the existing record if the event key is already present. `club_id` and `type` can be used to filter the data when users query the metrics from *mass comm* as the two fields are defined as primary keys in the `pk.fields`.

5

Experiments

5.1 Experiment Design

To demonstrate how switching to a distributed Event-Driven architecture would impact the overall performance, we designed several experiments for *mass comm* with *vg-monolith* being the baseline. To solely focus on the performance of *mass comm* and *vg-monolith*, we replaced the endpoint of notification delivery services (FCM, APN and Amazon SNS) with a dummy endpoint deployed by ourselves. In those experiments, we measure the overall process time to send certain amount of push notifications. Each experiment was run 20 times for both *vg-monolith* and *mass comm*. The average values were then used as the final results. The tests we ran are as follow:

- **Latency test.** How much time in milliseconds does it take for *mass comm* and *vg-monolith* to send only one push notification.

- **Scalability test.** How much time in milliseconds does it take for *mass comm* and *vg-monolith* to send multiple push notifications. This ranges from 100 push notifications to 1,000,000 push notifications.

5.2 Experiment Environment

In this section, we present the experiment environment, in which *vg-monolith* and *mass comm* are deployed respectively. The *vg-monolith* instance we used for this experiment was deployed on an AWS Elastic Kubernetes Service(EKS)[1] cluster. EKS offers great flexibility and simplicity for building and managing Kubernetes clusters on AWS EC2[2] instances. We deployed our *vg-monolith* instance inside a node group that consists of `t3a.large` EC2 instances. The detailed specification of `t3a.large` instances can be found in Table 5.1. And we allocated `1000m` CPU and `1000Mi` memory for the *vg-monolith* instance in the Kubernetes deployment configuration.

Processor type	AMD EPYC 7571
Clock speed (GHz)	2.5
Number of vCPU	2
Memory (GiB)	8
Network bandwidth (Gbps)	5

Table 5.1: Specification of AWS t3a.large EC2 instances.

For the lambda functions in *mass comm*, we assigned `1024MB` of memory to each one of them. According to the documentation(46)(47), there is no CPU configuration directly exposed. However, the amount of memory also determines the amount of virtual CPU available to a function. Yet, there is no clear information indicating the CPU/Memory ratio.

It is also important to list the detailed specification of the Kafka cluster as it is a core component of *mass comm*'s architecture. We deployed a MSK cluster of `kafka.m5.large` type and it consists of 3 brokers. Table 5.2 shows the specifications of type `m5.large`.

Processor type	Intel Xeon Platinum 8175
Clock speed (GHz)	3.1
Number of vCPU	2
Memory (GiB)	8
Network bandwidth (Gbps)	10

Table 5.2: Specification of AWS m5.large MSK instances.

It is apparent that the performance of the Opensearch instance will significantly influence the overall performance of *mass comm* since there are frequent communication between the lambda functions and the Opensearch instance. The specification of the Opensearch deployment, which is type `t3.small.search`, can be found in Table 5.3.

[1]https://aws.amazon.com/eks/
[2]https://aws.amazon.com/ec2/

Processor type	Intel Skylake E5 2686 v5
Clock speed (GHz)	3.1
Number of vCPU	2
Memory (GiB)	2
Network bandwidth (Gbps)	5

Table 5.3: Specification of AWS t3.small Opensearch instances.

5.3 Experiment Results

Figure 5.1 shows the average process time for *vg-monolith* and *mass comm* to send only one push notification, which is the latency test. In our first test run, the result from *mass comm* is astonishingly high, which is around 20,000 milliseconds. In contrast, the process time from *vg-monolith* is significantly lower, around 2,000 milliseconds. However, when running the test again immediately after the first run, the result looks satisfying, only around 1,000 milliseconds. Later we found out this is because of the cold start latency of AWS lambda(48). When AWS lambda service receives a request but there is no ready lambda execution environment ready, the service will first prepare one, which includes downloading the function code uploaded by users and allocating an environment with memory and runtime. Once the execution environment is ready, the user-provided function code will be executed. The preparation of execution environment is also called "cold start". After finishing the first execution, the execution environment will be preserved for a non-deterministic period of time. If another request arrives in this time window, the execution environment will be reused, hence skipping the preparation steps. This is called "warm start".

As a result, we sampled the process time for *mass comm* with cold start and without cold start. The execution time without cold start is nearly half the execution time of *vg-monolith*. Meanwhile, the cold start latency can heavily defect *mass comm*'s performance.

In Figure 5.2, the experiment result of sending multiple push notifications, the scalability test, is shown. The test results from *mass comm* are all sampled without cold start latency. We were only able to send out fewer than 100,000 push notifications from *vg-monolith* due to SQL errors occurring when attempting to send notifications to 100,000 users. These errors were caused by the SQL statements being too long.

While *vg-monolith* performed well when sending 100 and 1,000 push notifications, its

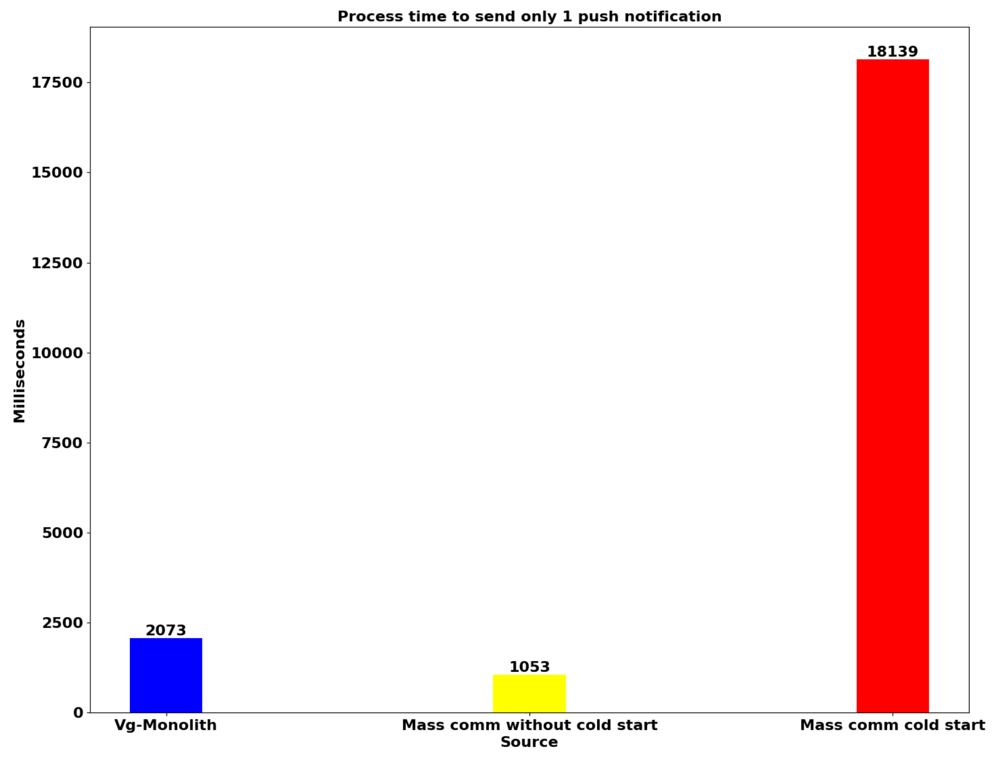


Figure 5.1: Average process time to send 1 push notification

performance degrades significantly when sending 10,000 push notifications. In contrast, *mass comm* only exhibits performance degradation when sending to 1,000,000 users. Furthermore, the performance of *mass comm* remains unaffected regardless of the number of notifications sent, as long as it is below 100,000.

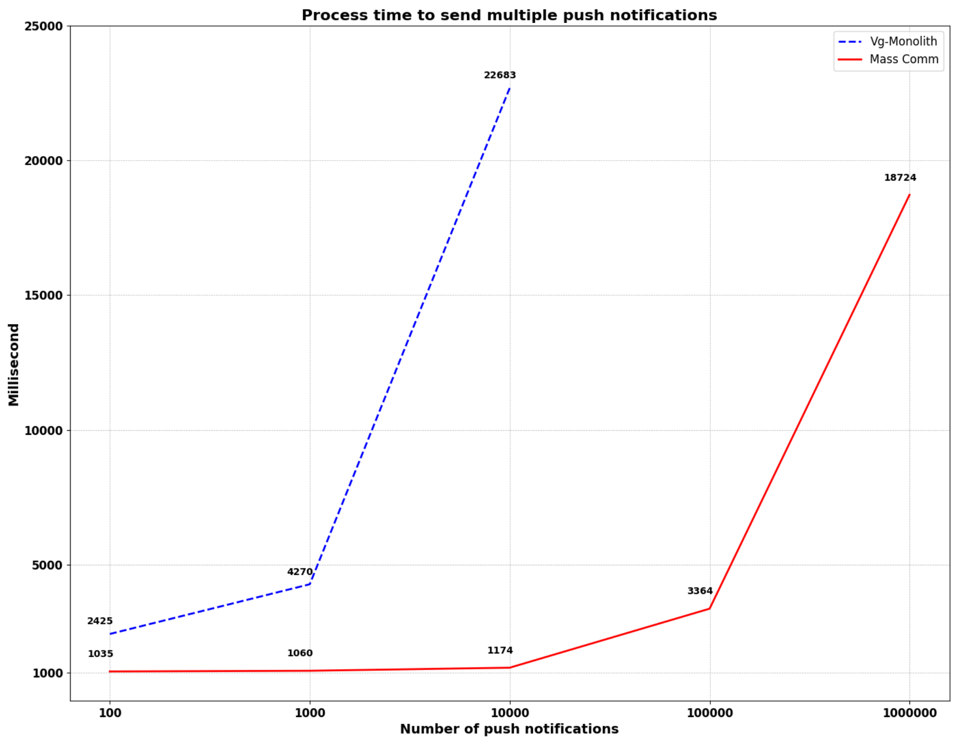


Figure 5.2: Average process time to send multiple push notification

6

Discussion

6.1 Reflection on Experiment Results

The result of the performance experiment is satisfying and meets the expectation. *Mass comm* shows better scalability in the experiment while *vg-monolith* is not even able to process such high volume of work. The process time to send one push notification from *mass comm* is surprisingly low. We expected *vg-monolith* would perform slightly better than *mass comm* in this test because everything happens in a shared memory context for *vg-monolith*. On the other hand, the architecture of *mass comm* is fully distributed. The cross-network communication is more frequent throughout the execution in *mass comm*, which could potentially harm the performance. However, the experiment result shows that this is not the case. And this stands as a strong testimony for the low-latency nature of EDA.

The results from the scalability tests also did not surprise us. *Mass comm* was able to stay uneffected when the compute load increased, due to the auto-scaling mechanism of AWS lambda(49). Additionally, we enabled message batching for each Lambda function. With this configuration, a single Lambda execution can process nearly 1,000 Kafka events, which explains why the execution time increased only marginally when the load increased from 100 to 10,000. However, when the compute load reached 1,000,000, *mass comm* experienced a significant delay. In-depth monitoring revealed that this delay was due to the limited Kafka event throughput of each Lambda function. If the producer of Kafka events is not generating events as quickly as the consumer processes them, the AWS Lambda service will not instantiate more execution environments, as the events are consumed without delay.

We believe that this bottleneck in event production is due to network latency and is not resolvable. This is supported by a small experiment we conducted (see Figure 6.1). When we sent the same 100 Kafka events, all serialized with an AVRO schema, to both a local Kafka cluster and our test MSK cluster, the process times were significantly different. And the process time of sending to MSK cluster matches the process time we observed in the lambda function logs.

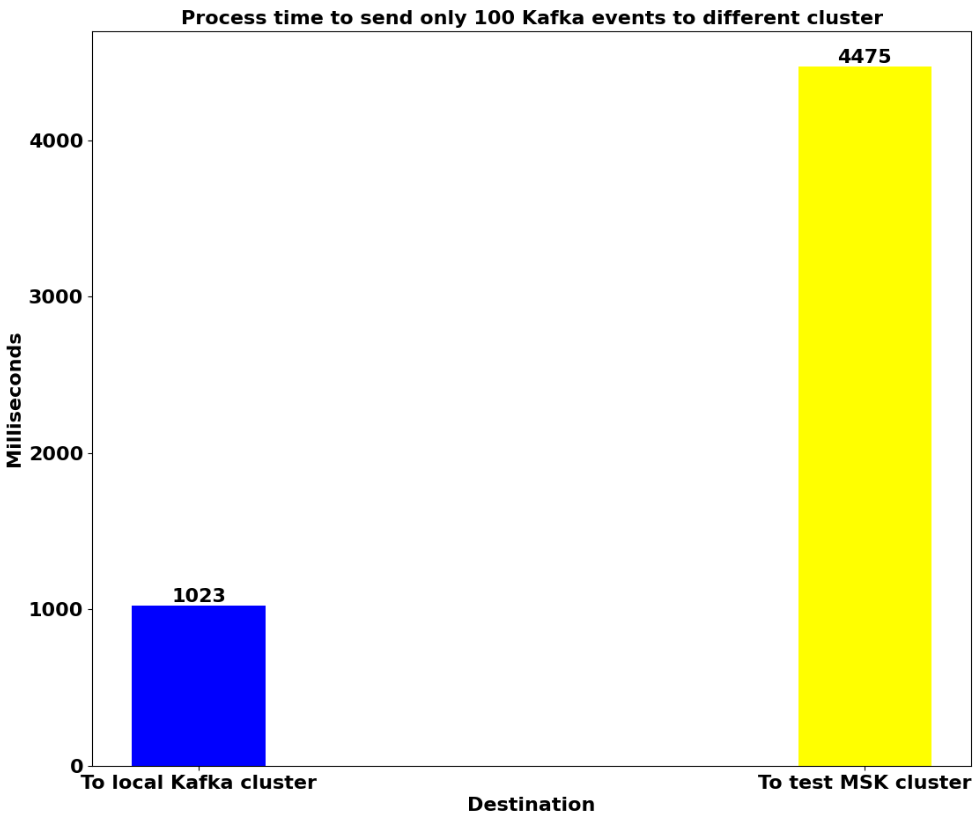


Figure 6.1: Average process time to send 100 Kafka events

6.2 Future Works

6.2.1 Impact of Lambda Function Cold Start

As shown in Figure 5.1, the cold start delay of lambda functions is more than 10 times higher than the actual execution time. This latency can drastically defect user experience as taking 18 seconds to send 1 push notification is not acceptable. To tackle this issue, we will have to explore the provisioned concurrency setting(49) offered by AWS. By setting the number of provisioned concurrency for a lambda function, AWS lambda service guarantees that there will always be the same number of lambda execution environments available for that function. Therefore, cold start latency is eliminated for certain amount of requests.

However, setting provisioned concurrency also limits the maximum concurrency for other lambda functions. This is because all the lambda functions deployed under the same AWS account share the same concurrency pool, this number is usually 1,000. When setting certain amount of concurrency for one function, for example, 400, all the other function can only use the remainder concurrency, which is only 600. As a result, the traffic pattern for each lambda function should be carefully studied before setting this number.

6.2.2 Limitations of ksqlDB

Originally, the Postgres database was not part of the whole architecture of *mass comm*. Data persistent can be handled by Kafka by adjusting its configuration(50) to permanently store the data in some certain topics. However, the Postgres database is still needed due to some limitations of ksqlDB.

The ideal implementation of the metrics feature is to make the queries directly from Metrics Handler lambda function to tables like `pushnotification_table_minute` in ksqlDB. The problem is, when querying metrics with one or more filters (`club_id` and `type`) missing, the data should be aggregated by using `GROUP BY` keyword because each record in the table is bounded by all the filters. Unfortunately, this is not possible because *pull queries* do not support `JOIN`, `PARTITION BY`, `GROUP BY` and `WINDOW` clauses(51). The first instinct to solve this problem is to use *push queries* instead of *pull queries*. But this would require setting up a HTTP streaming connection between Metrics Handler and the user because *push queries* can only be terminated manually, which will spawn tow more problems. Firstly, the timing of terminating the *push query* is uncertain and varies as the amount of data grows. The purpose of the Metrics Handler is to return a snapshot of the current metrics. Terminating the query immaturely would damage the integrity of the result. Secondly, even if the decision of termination is given to the users of *mass comm*, it is still not viable. As of the current date (March, 2024), AWS Lambda only supports returning stream responses when using Javascript(52). However, there are some certain python libraries from Virtuagym that have to be used. Another workaround for this issue we discovered requires to create two more ksql queries for each time unit metric. This was not accepted because it would become not scaleble running too many queries increases the workload of ksqlDB.

Another limitation of ksqlDB is its `WINDOWSTART` pseudo column. The original queries used to create ksql tables leverages the powerful window grouping feature(53) offered by ksqlDB. By doing so, there is no need to calculate the window start like `timestamp - timestamp % 60000` and do the grouping manually. Instead, it can be replaced by `WINDOW TUMBLING (SIZE 1 MINUTE)`. And this creates a pseudo column `WINDOWSTART`. However, the `WINDOWSTART` cannot be used in the `pk.fields` when creating the connector. See Figure 7.1 for the errors. Not having the starting timestamp of window will cause the records in the database table not grouped by the timestamp, which leads to fragmented data. Due to this reason, we have to manually calculate window start and group the data.

As a result, exploring other solutions like Apache Flink is considered one of the future plans if applicable.

6.2.3 Exploration of OpenSearch Serverless

AWS OpenSearch offers a serverless solution(54) that provides better scalability. By switching into serverless, the OpenSearch cluster will only consume the minimal resources when there is few or no incoming requests. This on-demand scaling can therefore enable pay-as-you-go pricing and potentially reduce financial costs. However, it was not adopted in the first release into the production. This is because we were not certain how the traffic pattern of *mass comm* will be. Since OpenSearch Serverless requires a minimal of 2 OpenSearch Compute Units (OCU), plus the amount of storage necessary is around 1 TB, the minimal cost of OpenSearch Serverless is 750 USD per month. While the provisioned OpenSearch can be cheaper (435 USD per month) even if using c6g.2xlarge (8 vCPUs, 16 GB memory) nodes. Hence, transitioning to OpenSearch Serverless can be considered once the traffic pattern is confirmed, and if OpenSearch Serverless demonstrates greater cost-effectiveness.

7

Conclusion

This thesis presented the Mass Communication System, which is a highly scalable, event-driven data processing system aims to provide reliable push notification and email delivery and real-time metrics. The author extend the prototype, which only includes the adaptation of Apache Kafka and ksqlDB, and conducted further experiments to validate the concreteness of the design.

In Chapter2, the underlying technologies of *mass comm* are introduced. In addition, we also give the rationals for choosing those technologies. *mass comm*'s Event-Driven Architecture design consists of the following components: Apache Kafka, a distributed, highly scalable message broker; Apache Avro, the message serialization technique that boost the overall performance; Schema Registry, a schema management component; Change Data Capture, a reliable solution for data synchronization; and ksqlDB, a powerful tool enables real-time stream data processing based on Kafka. Furthermore, there are additional technologies that, while not part of the Event-Driven Architecture (EDA), can enhance the performance and scalability of *mass comm*, such as Serverless Computing, OpenSearch, and Amazon SNS. Besides that, we illustrate the depth of our decision-making process by comparing the technologies we selected with alternative options. In Chapter 3, we provide insights on multiple related works to see how our work distinguish from others.

The detailed design and implementation of *mass comm* are shown, as well as how ksqlDB is utilized to process real-time stream data. And finally, experiments compares *vg-monolith*, which is the legacy system responsible for message delivery, and *mass comm* to demonstrate the performance impact of transitioning from a monolithic architecture to a distributed

Event-Driven Architecture. The overall performance of *mass comm* suppress *vg-monolith* not only in the latency test but also the scalability test. As a result, we can answer our research questions as follow:

- **Does *mass comm* maintain low latency as expected from an EDA?**

  The results of the latency test, which compares the process time to send out only 1 push notification, shows that the latency of *mass comm* is even lower than *vg-monolith*. Therefore, we conclude the low latency characteristic persist even though the execution is scattered into different components that communicate over the network.

- **Does *mass comm* possess high scalability, which is necessary to emit significant amount of messages, such as 100,000 and 1,000,000.**

  From the results of the scalability test, we can see that *mass comm* is highly scalable even when the workload is around 100,000. The performance degradation only appears when the workload reaches 1,000,000. Thus, we conclude that *mass comm* has high scalability.

However, *mass comm* is not perfect as we found certain limitations in ksqlDB. Besides that, we list future works that can further complete our research, such as in-depth study for AWS lambda cold start issue, searching for viable alternatives of ksqlDB and exploring AWS OpenSearch Serverless.

All in all, we deem the EDA to be an ideal solution in terms of distributed system design. The overall performance and scalability is significantly enhanced after adopting EDA as shown in our experiment results. The Mass Communication System represents a significant advancement in Virtuagym's infrastructure, showcasing the benefits of modern architectural patterns and technologies.

References

[5] Chris Richardson. **The Strangler Pattern**. iii, 3

[6] Mark Richards. *Software architecture patterns.* 2015. iii, 2, 5, 6, 7, 8

[15] Mitch Seymour. *Mastering Kafka Streams and ksqlDB.* O'Reilly Media, 2021. iii, 8, 9, 10, 13, 14

[21] **Schema Registry Overview | Confluent Documentation**. iii, 10, 11

[41] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. **Building LinkedIn's Real-time Activity Data Pipeline.** *IEEE Data Eng. Bull.*, **35**(2):33–45, 2012. iii, 17, 18, 20

[1] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. **Migrating from monolithic architecture to microservices: A Rapid Review**. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019. 1

[2] Chris Richardson. *Microservices patterns: with examples in Java.* Simon and Schuster, 2018. 1, 2

[3] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021. 1

[4] Rui Chen, Shanshan Li, and Zheng Li. **From Monolith to Microservices: A Dataflow-Driven Approach**. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017. 2

[7] Brenda M Michelson. **Event-driven architecture overview**. *Patricia Seybold Group*, **2**(12):10–1571, 2006. 5

[8] J Boner, D Farley, R Kuhn, and M Thompson. **The Reactive Manifesto**. 5

[9] Martin Kleppmann. **Designing Data-Intensive Applications**, 2019. 5, 6, 12

[10] Joel Spolsky. **The Perils of JavaSchools**, 12 2016. 5

[11] Vijay Srinivas Agneeswaran. *Big data analytics beyond hadoop: real-time applications with storm, spark, and more hadoop alternatives.* FT Press, 2014. 6

[12] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. **Trends in big data analytics**. *Journal of Parallel and Distributed Computing*, **74**(7):2561–2573, 2014. Special Issue on Perspectives on Parallel and Distributed Processing. 6

[13] **Apache Kafka**. 6

[14] Khin Me Me Thein. **Apache kafka: Next generation distributed messaging system**. *International Journal of Scientific Engineering and Technology Research*, **3**(47):9478–9483, 2014. 6

[16] Nishant Garg. *Apache kafka.* Packt Publishing Birmingham, UK, 2013. 9

[17] Vineet John and Xia Liu. **A survey of distributed message broker queues**. *arXiv preprint arXiv:1704.00411*, 2017. 9

[18] Aditya Eka Bagaskara, Setyorini Setyorini, and Aulia Arif Wardana. **Performance Analysis of Message Broker for Communication in Fog Computing**. In *2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 98–103, 2020. 9

[19] Deepak Vohra. **Practical Hadoop Ecosystem**. *Chapter in Apache Parquet*, 2016. 10

[20] **Apache Avro documentation**. 10

[22] Kazuaki Maeda. **Performance evaluation of object serialization libraries in XML, JSON and binary formats**. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182, 2012. 11

[23] Darshan M. Tank, Amit Ganatra, Y.P. Kosta, and C.K. Bhensdadia. **Speeding ETL Processing in Data Warehouses Using High-Performance Joins for Changed Data Capture (CDC)**. In *2010 International Conference on*

Advances in Recent Technologies in Communication and Computing, pages 365–368, 2010. 12

[24] Itamar Ankorion. **Change data capture efficient ETL for real-time bi**. *Information Management*, **15**(1):36, 2005. 12

[25] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. **All aboard the Databus! Linkedin's scalable consistent change data capture platform**. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery. 12

[26] **Change Data Capture (CDC): The complete introduction | Confluent**. 12

[27] Mitchell J. Eccles, David J. Evans, and Anthony J. Beaumont. **True Real-Time Change Data Capture with Web Service Database Encapsulation**. In *2010 6th World Congress on Services*, pages 128–131, 2010. 12

[28] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann, 1995. 12

[29] JIŘÍ NOVOTNÝ. **Debezium performance testing**. 13

[30] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. **A Survey of Distributed Data Stream Processing Frameworks**. *IEEE Access*, **7**:154300–154316, 2019. 13

[31] Confluent. **Tables - KSQLDB documentation**. 13

[32] **Flink vs Kafka Streams/ksqlDB: Comparing Stream Processing Tools**. 14

[33] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. **Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark**. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery. 14

[34] Brendan Burns. *Designing distributed systems: patterns and paradigms for scalable, reliable services.* " O'Reilly Media, Inc.", 2018. 14

[35] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. **Cloud Programming Simplified: A Berkeley View on Serverless Computing**, 2019. 14

[36] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. **Evaluation of Production Serverless Computing Environments**. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450, 2018. 14

[37] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015. 15

[38] **DB-Engines ranking**. 15

[39] Doina R. Zmaranda, Cristian I. Moisi, Cornelia A. Győrödi, Robert Ş. Győrödi, and Livia Bandici. **An Analysis of the Performance and Configuration Features of MySQL Document Store and Elasticsearch as an Alternative Backend in a Data Replication Solution**. *Applied Sciences*, **11**(24), 2021. 15

[40] Mustafa Ali AKCA, Tuncay Aydoğan, and Muhammer İlkuçar. **An analysis on the comparison of the performance and configuration features of big data tools Solr and Elasticsearch**. *International Journal of Intelligent Systems and Applications in Engineering*, **4**(Special Issue-1):8–12, 2016. 15

[42] Florian Fröse, Daniel Bauer, Daniel Pittner, and Sean Rooney. **Unifying Metadata for Stream and Batch Queries in a Cloud Service.** In *CDMS@ VLDB*, 2022. 19

[43] Adeyinka Akanbi. **ESTemd: A Distributed Processing Framework for Environmental Monitoring based on Apache Kafka Streaming Engine**. In *Proceedings of the 4th International Conference on Big Data Research*, ICBDR '20, page 18–25, New York, NY, USA, 2021. Association for Computing Machinery. 20

[44] Sudeshna Dutta, Adrián Miranda, and Pablo Arboleya. **Real-time Data Extraction, Transformation and Loading Process for LV Advanced Distribution Management Systems**. In *2023 IEEE Belgrade PowerTech*, pages 1–6, 2023. 21

[45] Adrian-Tiberiu Costin and Daniel Zinca. **Network traffic logger with real-time streaming and spoofing capabilities**. In *2022 International Symposium on Electronics and Telecommunications (ISETC)*, pages 1–4, 2022. 21

[46] **Memory and computing power - AWS Lambda**. 30

[47] **Troubleshooting Lambda configurations - AWS Lambda**. 30

[48] **Lambda execution environments - AWS Lambda**. 31

[49] **Understanding Lambda function scaling - AWS Lambda**. 35, 37

[50] **Apache Kafka**. 37

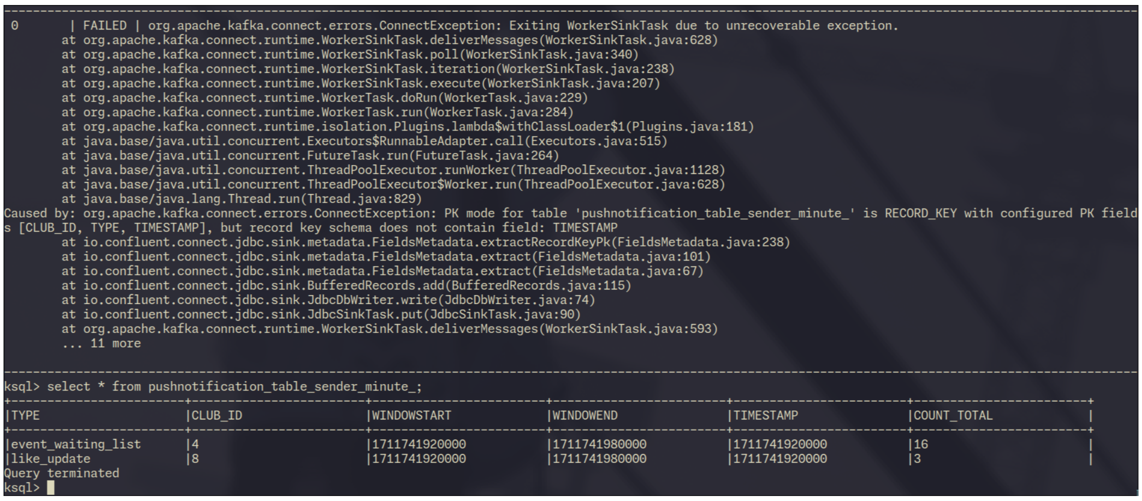
[51] Confluent. **SELECT (Pull query) - KSQLDB documentation**. 37

[52] **Introducing AWS Lambda response streaming | Amazon Web Services**, 1 2024. 37

[53] Confluent. **Time and Windows in KsqlDB - KsqlDB documentation**. 38

[54] **What is Amazon OpenSearch Serverless? (01:35)**. 38

Appendix



```
----------------------------------------------------------------------------------------------------------------------------------------------
 0       | FAILED | org.apache.kafka.connect.errors.ConnectException: Exiting WorkerSinkTask due to unrecoverable exception.
        at org.apache.kafka.connect.runtime.WorkerSinkTask.deliverMessages(WorkerSinkTask.java:628)
        at org.apache.kafka.connect.runtime.WorkerSinkTask.poll(WorkerSinkTask.java:340)
        at org.apache.kafka.connect.runtime.WorkerSinkTask.iteration(WorkerSinkTask.java:238)
        at org.apache.kafka.connect.runtime.WorkerSinkTask.execute(WorkerSinkTask.java:207)
        at org.apache.kafka.connect.runtime.WorkerTask.doRun(WorkerTask.java:229)
        at org.apache.kafka.connect.runtime.WorkerTask.run(WorkerTask.java:284)
        at org.apache.kafka.connect.runtime.isolation.Plugins.lambda$withClassLoader$1(Plugins.java:181)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
        at java.base/java.lang.Thread.run(Thread.java:829)
Caused by: org.apache.kafka.connect.errors.ConnectException: PK mode for table 'pushnotification_table_sender_minute_' is RECORD_KEY with configured PK field
s [CLUB_ID, TYPE, TIMESTAMP], but record key schema does not contain field: TIMESTAMP
        at io.confluent.connect.jdbc.sink.metadata.FieldsMetadata.extractRecordKeyPk(FieldsMetadata.java:238)
        at io.confluent.connect.jdbc.sink.metadata.FieldsMetadata.extract(FieldsMetadata.java:101)
        at io.confluent.connect.jdbc.sink.metadata.FieldsMetadata.extract(FieldsMetadata.java:67)
        at io.confluent.connect.jdbc.sink.BufferedRecords.add(BufferedRecords.java:115)
        at io.confluent.connect.jdbc.sink.JdbcDbWriter.write(JdbcDbWriter.java:74)
        at io.confluent.connect.jdbc.sink.JdbcSinkTask.put(JdbcSinkTask.java:90)
        at org.apache.kafka.connect.runtime.WorkerSinkTask.deliverMessages(WorkerSinkTask.java:593)
        ... 11 more

----------------------------------------------------------------------------------------------------------------------------------------------
ksql> select * from pushnotification_table_sender_minute_;
+------------------------+------------------------+------------------------+------------------------+------------------------+------------------------+
|TYPE                    |CLUB_ID                 |WINDOWSTART             |WINDOWEND               |TIMESTAMP               |COUNT_TOTAL             |
+------------------------+------------------------+------------------------+------------------------+------------------------+------------------------+
|event_waiting_list      |4                       |1711741920000           |1711741980000           |1711741920000           |16                      |
|like_update             |8                       |1711741920000           |1711741980000           |1711741920000           |3                       |
Query terminated
ksql> 
```

Figure 7.1: The connector error when using WINDOWSTART. The TIMESTAMP is created by SELECT ... WINDOWSTART as TIMESTAMP ....