Implementation & architecture of a cloud-based data analytics pipeline

John Reuterswärd
Abstract

Implementation & architecture of a cloud-based data analytics pipeline

John Reuterswärd

Organizations can gain deeper insight into the usage of their products and services by analyzing data from events as they happen in real-time. Examples of such data include user login attempts, feature usage, sign-ups, client logs as well as other domain and application specific events. The high frequency at which these types of events are generated combined with the large volume of data they will occupy once stored provides many interesting challenges. Most importantly, it is difficult to predict what type of analytics data will provide useful insights in the future. This master thesis documents the architecture of a data analytics pipeline based on the principles of decoupled components to meet this goal. It is shown that by extending the concepts of the publish & subscribe pattern and by leveraging cloud-based services the development of such a system is achievable even for smaller organizations.
Contents

Glossary 4

1 Introduction 5

2 Background 7
   2.1 About DigiExam 9

3 Related work 10

4 The publish/subscribe pattern 13

5 Evaluation criteria 15
   5.1 Flexibility 16
   5.2 Scalability 17
   5.3 Robustness 17

6 Core components 18
   6.1 BigQuery 18
      6.1.1 Comparing BigQuery with RDBMS 19
      6.1.2 Comparing BigQuery with MapReduce 19
   6.2 Cloud Pub/Sub 20
      6.2.1 Comparing Cloud Pub/Sub with Apache Kafka 20
   6.3 Go 20

7 Implementation & Architecture 22
   7.1 Topics 23
   7.2 Producer 24
   7.3 Consumer 26
   7.4 Long-term metrics 27
   7.5 Real-time metrics 30
   7.6 Management tools 31
8 Results
8.1 Flexibility ........................................... 33
8.2 Scalability ........................................... 35
8.3 Robustness ........................................... 36

9 Discussion & conclusion .................................. 38
9.1 Decoupled design ....................................... 38
9.2 Familiar query language ................................ 38
9.3 Tools and plumbing .................................... 39
9.4 Cloud-based architecture ............................... 39
9.5 Future work ........................................... 40

List of Figures ........................................... 41
List of Tables ........................................... 42
List of listings .......................................... 43
Bibliography ............................................. 44
Acknowledgments

My deepest gratitude goes to my supervisor Robin Andersson and my reviewer Matteo Magnani. I want to thank Robin for giving me the opportunity to write my master thesis at DigiExam, for the support he has shown me and for making sure I had everything I needed to complete my work. I want to thank my reviewer Matteo Magnani for his guidance, valuable expertise and his time. I also want to thank Matteo for encouraging me to push on and pursue my ideas. Lastly, I want to thank Kenneth, Kenan, Nima and Johan at DigiExam for taking an interest in my work and for being a great team to work in.
Glossary

**Consumer** A consumer is a program that consumes messages from a subscription to a topic.

**Long-term data** Data from messages that is persisted to long-term storage in batches for deeper insight analysis.

**Message** A message is an individual packet of data that consists of a payload and optional meta-data. A message conforms to a topic schema.

**Producer** A producer is a program that publishes messages to a topic.

**Pull subscription** A pull subscription is a consumer subscription where the consumer periodically pulls a subscription for new messages from a topic.

**Push subscription** A push subscription is a consumer subscription where the consumer automatically receives new messages after they have been published to a topic.

**Real-time data** Data from messages that can be analyzed and visualized within seconds rather than minutes, hours or days.

**Subscription** A subscription is a consumption feed for new messages from a particular topic.

**Topic** A topic represents a logical channel of communication to which messages are grouped by and sent to. A topic has a schema of zero or more properties.
Chapter 1

Introduction

Many organizations depend on insights from data to help them in their decision making because it enables them to stay competitive. Deciding what data to store and how to store it is seldom an easy decision. An even harder challenge is turning the data into something useful. In this context, analytics is the umbrella term that covers both how and what data to store as well as what to do with it. Through data-driven analytics, organizations discover trends, usage patterns and many other types of useful information. The concept of data warehousing is a closely related term that refers specifically to how data is extracted, prepared, stored and made available [16, 19].

Increasingly, many organizations are interested in the ability to analyze data from events in a real-time capacity which brings many new challenges. For web applications with many thousands of concurrent users, user activity data is often of particular interest. Activity data helps organizations understand how their users use a service. Storing meta-data that describes these events also help organizations identify problems and mitigate potential security issues more quickly.

There are many types of analytics data that are of interest among many organizations. For example, this can include things like user login attempts, roll-out of client updates, error logs and web clicks to name a few. Recent developments in database technologies and the adoption of cloud-based services have made the development and cost associated with such systems feasible and practical [24].

The challenge of capturing and storing activity data in a real-time fashion lies in handling the often large volumes of both structured and unstructured data that is generated at a high frequency. The data must be handled in a fast, reliable and uniform way. Furthermore, the data needs to be accessible to the people that intend to use it [10]. Depending on the type of data
metric, both real-time and long-term analysis might be of interest.

It is difficult to predict how data will be used in the future and easy to make premature assumptions regarding what metrics will be of most interest. Because of this, many data analytics system fail to meet the demands of the customer [15].

A useful abstraction is to treat an analytics system as a pool of data through which interested parties can get the data they want and use it as they wish. The idea of a data pipeline is closely related. A pipeline is defined as a set of stages through which data is processed. Analogous to a water pipe system, such an architecture should support the ability for users within an organization to connect to a particular point and get to the data they want [3, 10, 24]. This master thesis documents the design and implementation of the architecture for such a system.
Chapter 2

Background

When building analytics systems, many organizations use what is known as an ETL system, short for Extract, Transform, Load [16]. The purpose of an ETL system is to bring data from many different sources together in the same place. Typically, the sources contain heterogeneous data which is extracted from databases or files across different machines. Data is then transformed according to a specification thus making the heterogeneous data homogeneous. Lastly, the data is loaded into a persistent storage. Most ETL systems are batch oriented and commonly run on a fixed schedule [16].

Batch-oriented systems like ETL are a good fit when the goal is to consolidate large volumes of data in order to generate reports, perform long-running aggregation tasks or simply to get all the data in one place. However, the batch oriented model of an ETL-like system is not a good fit when dealing with Real-time data. In many cases the implementation of an ETL-like system is tightly coupled with the data itself which makes the system less flexible [24].

Analytics data is data that typically describes an event or activity. As an example, consider a web application with a login screen as seen in Figure 2.1. When a user types in her information and submits the form the server receives the request and checks if the provided information was correct. Regardless of the outcome of the login we might want to make a note of when this happened, who tried to login and what happened. By storing and analyzing these events we can extract information that could be useful in several ways.

For example, we could identify the time of day in which people log in and use this information to scale server infrastructure accordingly. It could be used to analyze if users struggle to remember their passwords. Perhaps there is some aspect of the login page that could be simplified or further
explained to the users. If there is an explosive increase in the number of login attempts per second to a particular user account it may be a cause for concern as someone other than the account holder might be trying to gain access. When the users clicks the log in button we send off the following information as soon as the outcome is determined:

```json
{
   "action": "user-login",
   "user": "jane@bigco.com",
   "success": true,
   "timestamp": "2015-06-11 21:39"
}
```

There are many other user initiated events that are of interest. For instance, we may be interested in the amount of time users spend performing a particular task. We might also wish to analyze the level of usage of a particular feature. Perhaps we wish to identify if there is a particular group of users that use the feature more than others. The following JSON snippet shows how a feature usage could be represented:

```json
{
   "action": "invited-user-by-email",
   "user": 7824819,
   "feature": "user-invite",
   "invitedUsers": [7824817, 187212, 298425],
   "timestamp": "2015-04-12 13:40"
}
```

In the above example the `action` property shows what particular action has been used. In this case we can imagine a service where users can invite other users in different ways and in this instance email invitation was used. The `feature` property represents a higher-level grouping of the action so that we can later find all the related actions. The `invitedUsers` property shows the user IDs of the invited users. This relationship could later help us identify if the new users also invite other users using email.

We can think of the JSON snippets as representations of messages that are used to communicate between different systems. In the first example, we communicate messages that are relevant to any system that is interested in learning about user login attempts. A suitable grouping for these messages would then be to group them by a topic, similar to a discussion or conversation. We can extend the notion of a topic to include a schema that specifies which properties a message should include. The idea of messages and topics is a central concept in a publish/subscribe system which is explored in Chapter 4.
In application with many concurrent users capturing these types of events result in a very high *velocity* of data as a single user might trigger several of these events during a session. Ideally, we wish to send the data away as soon as possible while it is still relevant. In time the *volume* of data that is generated by these messages will require scalable architecture for moving and storing the data. Finally, we want to have the ability to analyze and measure any kind of event which means that there will be a big *variety* of the data that is generated. *Velocity, volume and variety* are among the defining characteristics of the concept of *Big Data*. While the term *Big Data* has no clear definition it has come to represent the challenges many organization are facing in building scalable data-intensive architectures [9].

There has been a lot of innovation in this space in recent years as more and more organizations are figuring out new ways to build scalable data systems. The advent of cloud computing has in many ways made the development of the kinds of systems needed to cope with these problems much simpler as databases and other systems can be provided as services. In many cloud-based services the implementation and configuration details are abstracted from the user. This allows smaller organizations to compete with larger organization that have more resources.

### 2.1 About DigiExam

This master thesis was conducted at DigiExam¹ which is an educational technology company. DigiExam provides a software platform as a service for the creation, examination and grading of exams for schools. DigiExam uses the Google Cloud Platform (GCP) which is public cloud computing platform [13]. DigiExam utilizes many of the GCP services and products such as App Engine and Compute Engine. The core backend software is written in Go [27].

¹[https://www.digiexam.se/]
Chapter 3

Related work

The Evolution of Publish/Subscribe Communication Systems by Baldoni, Contenti and Virgillito provides a comprehensive definition of the most important concepts of a publish/subscribe (pub/sub) system [4]. In the article, the authors classify two main classes of pub/sub systems: topic-based and content-based. In a topic-based system, processes exchange information through a set of predefined topics which represent logical channels. The article goes into detail to explain how a pub/sub system works in general and makes the case that pub/sub systems are relevant to many different research communities including those of databases, distributed systems and software engineering [4]. The publish/subscribe pattern is further explored in Chapter 4.

Goodhope and colleague present the work and challenges of building a real-time activity data pipeline at the social-networking site LinkedIn [10]. The authors outline the design and engineering problems they faced when migrating from an existing batch-oriented system to a real-time publish/subscribe system. The paper centers around the design and implementation of the yafka messaging system [2].

The development of Kafka is motivated by showing how data analytics drive many aspects of LinkedIn. As an example, user activity data is inserted into machine learning models to predict relevant content for the users. Continuous streams of user activity data also form the basis of many security measures at LinkedIn as real-time monitoring enables engineers to track and prevent abuse and fraud more quickly. One of the stated goals of the project was to enable different teams within the organization to integrate their own tools with the pipeline and subscribe to the feeds they are interested in. The biggest challenges of the project was handling all the diverse data that fed into the pipeline. The authors share their experience in using different
serialization formats for encoding data. They also explain how they handled changes to the messaging schema [10, 25].

On the topic of diverse data, Lee et al. aim to explain and simplify how the data that go into a data pipeline can be collected, structured and unified in the first place [17]. The authors argue that while business intelligence and data warehousing have been around for decades, the field of data analytics has entered a new era facing new challenges. In the paper, Lee et al. document their work towards streamlined log collection and data analysis which is used in production at the social-networking company Twitter.

The paper explains how their approach to logging evolved through several iterations. The authors identified user sessions as a common entry point for many queries in the domain of web applications. They present a novel approach to logging which they call session sequences in which log events are grouped based on user sessions. The paper contains many practical ideas. As an example, the authors document what they believe is the most effective way to normalize different data formats and data types as well as how to deal with flat and nested data [17].

Bae et al. has published a detailed post on the Netflix Tech Blog which discusses the case for building a data pipeline at the video-streaming company Netflix [3]. The authors outline the overall architecture of the system and how it is deployed on Amazon Web Services (AWS)\(^1\). Netflix developed a central collection system called Suro\(^2\) which dispatches incoming data to a batch processing pipeline using Apache Hadoop, and a real-time pipeline that uses Apache Kafka [1]. The authors motivate the development of the data pipeline at Netflix by explaining how real-time pipelines enables operational insights and instant feedback on changes to their services. A notable design goal of the system is the support for arbitrary data formats which would allow developers to use their own serialization code. The post is especially of interest because the implementation relies entirely on a cloud based infrastructure which is becoming increasingly important [3].

In a post on the Twitter engineering blog, Ed Solovey shows how they built an analytics system that is capable of handling five billion sessions each day with real-time constraints [26]. The architecture consists of decoupled components which use asynchronous messaging using Apache Kafka to communicate. Events are compressed and sent in batches from the client devices to a web server written in Go which enqueues events to Kafka. The system relies on using MapReduce for batch computation of events which

\(^1\)https://aws.amazon.com/
\(^2\)https://github.com/Netflix/suro
are then stored in a Apache Cassandra cluster [26]. The post provides useful comments on important architectural principles especially in regard to failure handling.

Cloud computing plays an important role for the modern enterprise. The importance of scalability, costs and zero-downtime are some of the topics covered in the book The Cloud at Your Service by Rosenberg and Mateos which provides lots of practical considerations regarding cloud-based computing and what it means for developers, managers and organizations as a whole [22].

The concept and usefulness of a data pipeline also feeds into the larger discussion of data warehousing and data analytics [24]. As such, many practical considerations and ideas have been extensively documented in books and papers. The Data Warehouse Toolkit by Ralph Kimball and Margy Ross covers many of the most important concepts [16]. The implementation of data pipelines is also closely associated with the domain of Big Data, meaning high volume, high velocity and high variety of data. This new era of big data and analytics is discussed further by Wixom et al. who outline the emerging market of data analytics at a larger scale [30].
Chapter 4

The publish/subscribe pattern

The publish/subscribe pattern is a message oriented communication paradigm which defines two types of actors, publishers and subscribers. The communication link between the publishers and the subscribers is a topic. A topic is logical in some contexts topics are referred to as channels. A publisher is an actor that publishes messages to topic. A subscriber is an actor that subscribes to messages from a topic. In a publish/subscribe system publishers sends messages to a topic from which a subscriber can get them [4, 14, 28].

To assess the appropriateness of using a publish/subscribe pattern for a data pipeline, it is important to review some alternative methods. Consider a scenario in which a web service stores user login attempts and several different consumers are interested in that particular data. What each consumer intends to use the data for is not important. In this particular scenario we can imagine that at least one of the consumers wish to show the login attempts as they occur within a reasonable delay and at least one of the other consumers only wishes to store long-term records of all attempts in a database.

One alternative pattern is based on polling. In this scenario each of the consumers periodically polls the web service for updates. This pattern works well for the real-time consumer as the polling rate can be tuned to an acceptable delay. If the web service has the ability to cache login attempts for some time this approach also works well for the long-term consumers as it could periodically poll the web service and get cached data in batches. However, if there are no new updates the consumers waste bandwidth and computing power checking for messages. If we extend the scenario to include login attempts from another service each consumer must poll every web service.
Another alternative is for the web service to broadcast the attempts as they occur. This approach is inherently inflexible since the web service is now responsible for communicating with each of the consumers. The web service also needs keep in sync with any changes made to any of the consumers. Furthermore, the web service either needs to know the rate at which each consumer wishes to obtain the data or naively deliver the data to all consumers even if they end up ignoring most of the data.

The problem with both of these approaches is that the consumers and producers are *coupled*. With the polling approach the consumers need to keep up to date with any changes to the web service and in the broadcasting approach the web service needs to keep up to date with changes to the consumers.

Translating the scenario to a publish/subscribe system, we can define a topic to which the web services publish login attempts. Each consumer then has its own subscription to that topic. This effectively decouples the web clients from the consumers and lets us treat them as independent systems. This system is inherently more flexible as the web services only care about the existence of a particular topic and each consumer has its own subscription. Compared with the polling and broadcast approach the use of a topic as a logical channel of communication enables *late-binding* as the web service and the consumers do not have to know of each other [6, 14, 20].
Chapter 5

Evaluation criteria

This chapter outlines the evaluation criteria which will be used to evaluate the implementation and architecture. The criteria outlined in this chapter also formulate the requirements of the implementation. The evaluation criteria is divided into three categories: flexibility, scalability and robustness.

There are several important considerations that are relevant to the design of an analytics pipeline. One very important consideration is how it will be used by developers. Integrating the pipeline with existing and future systems and applications is perhaps one of the hardest things to get right. Is the design of the API intuitive and is the functionality and intent clearly visible and easily understood? Ideally, integration with existing applications should be frictionless and straightforward. One could argue that if using the analytics pipeline gets in the way of implementing new features or fixing bugs there is a good chance that it would not be used at all as the drawbacks outweigh the benefits.

Integration to the pipeline should be platform and language independent. One of the main advantages of the concept of an analytics pipeline is the ability for anyone to get to the data they want in an easy way. It would severely limit the flexibility and usefulness of the system if integration were somehow language or platform dependent. To avoid this it is important to consider protocols and data formats which are both platform and language agnostic. Finally, the analytics pipeline needs to have a security mechanism that prevents unauthorized access and provides transport security of data while in transit.

Another important aspect is how the analytics pipeline will adapt to heterogeneous data. One can imagine a situation where the schema associated with a topic has been revised and different producers are sending event data containing different fields to the same topic. To solve this problem, each
topic in the analytics pipeline should have a version associated with it.

Scalability is another important concern. What happens when the number of events that need to be handled rapidly increases? In many cases the producers need to be able to publish events with a relatively low latency because they also need to respond to user requests. When the amount of traffic becomes too large for a single server instance to handle, the analytics pipeline should be able to scale out as new instances are started. Another scalability concern is whether the system will remain manageable when the number of topics, producers and consumer has increased. Managing a handful of topics, producers and consumers is relatively easy, but what happens when we have hundreds?

Insuring fault-tolerance is also an important aspect. For example, what happens when a producer starts sending erroneous data? What strategy should be used when a consumer receives data that contains errors? Furthermore, network failures and changes in infrastructure may disrupt the system. As in any distributed system it is inevitable that at some point one or more components will go down. It is important to take these aspects into account when designing the system.

5.1 Flexibility

The flexibility criteria primarily refer to the ease of use of the analytics pipeline. Developers will interface with the analytics pipeline through an API which handles authentication, serialization and preparation of messages. Management of topics, producers and consumers should be straightforward and accessible. This is also a scalability requirement. The data format used should be flexible and be able to encapsulate a variety of topics. Once data is persisted to disk there should be a flexible way to both do ad-hoc queries and to run more complex jobs. The requirements of the flexibility criteria are as follows:

1. Integration with the analytics pipeline should be straight-forward and simple.
2. The API exposed to the developer should have a minimal footprint when used in the code.
3. The analytics pipeline should be platform and language agnostic.
4. It should be easy to manage topics, producers and consumers.
5. The data format should not impose any unnecessary restrictions on which types of events can be described.
6. It should be easy to ask questions and get answers from the data that is persisted.

5.2 Scalability

The scalability criteria concern the extent to which the analytics pipeline will be able to maintain performance in the face of higher volume of traffic. As the traffic increases more events will be sent into the pipeline. Furthermore, it is crucial that topics and subscriptions remain maintainable as more are added. The server to server latency for real-time data must be as low as possible. More importantly, the latency must be predictable and consistent. Over time, the volume of data stored will become very large. For the system to be future-proof the volume of data should not affect the ability to query and maintain it. The requirements for the scalability criteria are as follows:

1. The analytics pipeline should be able to scale as the load increases.
2. Topics and subscriptions should remain maintainable as more are added.
3. Real-time data should have sub-second latency between servers within the analytics pipeline.
4. Long-term data should remain easily accessible and workable even when million of records have been stored.

5.3 Robustness

The robustness criteria concern the ability of the analytics pipeline to handle failure. This includes system failures, network failures, loss of service availability as well as software errors. Furthermore, the analytics pipeline should be fault-tolerant in terms of handling erroneous data. The robustness criteria also concerns message delivery guarantees. To the extent possible, event messages published by a producer should be delivered to a subscribing consumer exactly once. The requirements for the robustness criteria are as follows:

1. A system or network failure should not affect the system as a whole.
2. Loss of service availability should be handled gracefully and not affect the rest of the system.
3. Erroneous data should be explicitly handled and logged.
4. Event messages should be delivered exactly once.
Chapter 6

Core components

This chapter presents the core components in terms of software, services and languages that were used in the implementation of the data analytics pipeline. The selection of these components was determined by evaluating the use-case and features they provide and how they compare to components that where outlined in Chapter 3 and are commonly used in these types of systems. It was also important to consider compatibility with existing infrastructure. As mentioned in Chapter 2 existing infrastructure at DigiExam runs on Google Cloud Platform. For the purpose of this master thesis project it was desirable to choose components that could be setup and run on GCP and that could be maintained by a handful of people. While this limits the choice of components to some degree, it also presents an opportunity to explore the managed services that GCP provide.

6.1 BigQuery

To be able to store and query massive amounts of data there was need for a database solution that is both flexible and scalable. BigQuery is a service for storing and querying very large datasets. BigQuery is the externalization of Dremel, a system developed at Google that has been used extensively since its inception in 2006. Use cases for Dremel at Google include analysis of crawled web documents, disk I/O statistics, spam analysis and resource management [23]. In the paper *Dremel: Interactive Analysis of Web-Scale Datasets*, Melnik and colleagues describe the implementation and architecture of Dremel [21].

Dremel uses a column-oriented storage system which separates the column values of a record and stores each value on a different storage volume. Columnar stores have several advantages over row-oriented stores for certain types of use-cases. Because different values of the same column are likely to be similar, a high compression ratio can be achieved more easily.
Disk I/O can also be reduced since scans only need to read a subset of the columns in the data [8]. Queries are dispatched from a root server down to intermediate servers which rewrite the queries for each level of the tree. Ultimately, the queries reach the leaf servers which scan a partition of the data in parallel. The intermediate servers then aggregate the result from the leaf servers to complete the result. BigQuery is a fully-managed Dremel system as a service. The storage and management of servers is transparent to the developer. The main interface to BigQuery is a REST API which is used to manage datasets, run queries and insert data [5, 11].

6.1.1 Comparing BigQuery with RDBMS

BigQuery uses a SQL-like query language and has the concepts of tables, columns and rows. There is however a big difference in how BigQuery works compared to a traditional relational database management system (RDBMS) like MySQL. Updates to individual records are inefficient in column-oriented stores. In BigQuery individual updates are not possible meaning that stored data is effectively immutable [23]. Compared to many RDBMS and NoSQL data stores, BigQuery only supports a small set of data types: string, integer, float, boolean, time and record which may contain one or more of the others.

In BigQuery, query execution time is usually in the order of seconds. This is true even for simple queries that involve relatively small datasets consisting of just a thousand records. In a RDBMS such queries would typically have an execution time in the order of milliseconds. This makes BigQuery impractical for use in applications where the majority of queries only scans a relatively small set of records or user facing applications where latency is critical. The advantage of BigQuery comes from its ability to maintain roughly the same execution time for datasets containing billions of records [5, 21, 29]. This makes BigQuery a good choice for storing large volumes of immutable data such as logs, metrics and other forms of analytics type data.

6.1.2 Comparing BigQuery with MapReduce

MapReduce is a programming model for processing and generating large datasets [7]. Apache Hadoop is a popular MapReduce implementation [1]. As mentioned in Chapter 3, Hadoop and related technologies are commonly used as a core component in a data analytics pipeline. Similar to BigQuery, the purpose of MapReduce is to facilitate processing of very large datasets. The advantage of the MapReduce model is that it facilitates long-running batch processing jobs that generate a lot of data. This is in contrast with BigQuery which is optimized for interactive processing jobs that usually generate a small result set [7, 21].

19
Because the MapReduce programming model is fairly low-level it is possible to do complex processing logic which would not be as easily achieved using the SQL-like language of BigQuery. Ultimately, MapReduce is better suited for problems that involve large volumes of unstructured data with large result sets, whereas BigQuery is better suited for problems that involve large volumes of structured data where the result set is relatively small [23, 29].

6.2 Cloud Pub/Sub

In order to send messages between different parts in the analytics pipeline a robust and scalable message based communication is needed. Cloud Pub/Sub\(^1\) is a cloud-based asynchronous messaging service [12]. It facilitates many-to-many messaging between computers. Similarly to BigQuery, Cloud Pub/Sub is provided as a fully-managed service. This means that the servers and infrastructure required to run is included in the service and completely transparent to the developer. Access to Cloud Pub/Sub is managed using an Access Control List (ACL) which handles permissions for topics and subscriptions. Cloud Pub/Sub makes no guarantees that messages will be delivered in the order in which they where sent.

6.2.1 Comparing Cloud Pub/Sub with Apache Kafka

As mentioned in Chapter 3, the Apache Kafka messaging system is a component commonly found in large-scale data analytics systems. Even though the feature-set of Kafka is much larger than Cloud Pub/Sub the two technologies share important similarities. Both Kafka and Cloud Pub/Sub offer durability as messages are persisted to disk and replicated across the network. Both Kafka and Cloud Pub/Sub guarantee at least once delivery [2, 12].

6.3 Go

In order to glue the different systems of the analytics pipeline together a programming language is needed. Practically any language could have been used here as the technologies are language agnostic. Go is a statically-typed, concurrent and imperative general-purpose programming language [27]. Go introduces the concept of goroutines which are light-weight processes. Goroutines communicate and share data using channels which can hold arbitrary data.

Listing 1 shows a concurrent Go program. Starting in the main function, we create a channel for boolean values named done. The main function starts

---

\(^1\) At the time of this writing Cloud Pub/Sub was in beta.
work in a separate goroutine by using the go keyword. Now, both main and
work run concurrently. When work is done, it sends a value on the shared
done channel to signal completion. The main function will wait on the last
line until it receives on the channel. Note that the actual value sent on the
channel, true, was discarded. This is because the channel was only used
to synchronize the state of the program. In another program the channel
could be used to return any errors or the result of a costly computation or
database query.

```go
1 func main() {
2     done := make(chan bool)
3     go work(done)
4     fmt.Println("Hello from main")
5     <-done
6 }
7
8 func work(done chan bool) {
9     fmt.Println("Hello from work")
10    done <- true
11 }
```

Listing 1: A concurrent Go program.
Chapter 7

Implementation & Architecture

The main architecture of the analytics pipeline is shown in Figure 7.1. The figure shows all the components of the implementation. Starting from the left, HTTP requests from the clients reach the application servers. The application servers handle the request, prepare an event message and publish to one or more topics. For example, a client request to login would trigger the creation of a user login event which is sent to the user login topic. The application servers are the producers of the analytics pipeline.

Event messages are persisted and replicated by the Cloud Pub/Sub service represented at the center of Figure 7.1. The service automatically issues a request to all subscriptions with a specified push endpoint. An external ACL service manages access to topics and subscriptions and is shown at the

Figure 7.1: Architecture of the data analytics pipeline.
The analytics servers shown on the right side of Figure 7.1 maintain subscriptions to various topics. Analytics server which have a push subscription will automatically receive new messages from Cloud Pub/Sub. Push messages are sent over HTTPS in a POST request to the subscribing server. An analytics real-time server, depicted at the bottom right corner of Figure 7.1 pushes events to a web-based dashboard. Long-term data is stored in BigQuery. The analytics server for long-term data periodically pulls a subscription for new messages and inserts them in batches. The analytics servers are examples of consumers of the analytics pipeline.

### 7.1 Topics

A topic is a logical channel of communication which is identified by its name. A topic has an associated schema consisting of zero or more fields. Every field has a name and a specified data type. A topic schema may not contain any nested fields. Topics are defined in Go packages which can be shared between applications. As an example, the schema associated with the `userlogin` topic is shown in Table 7.1. The associated schema for every topic is important for two reasons. The format of the data that a consumer will receive is static and specified. Furthermore, for every topic there exists a corresponding BigQuery table with a 1:1 mapping of the fields in the schema.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exist</td>
<td>boolean</td>
<td>True if the user was found</td>
</tr>
<tr>
<td>correctPassword</td>
<td>boolean</td>
<td>True if password was correct</td>
</tr>
<tr>
<td>disabled</td>
<td>boolean</td>
<td>False unless the user account is disabled</td>
</tr>
<tr>
<td>email</td>
<td>string</td>
<td>Email address</td>
</tr>
<tr>
<td>remoteAddr</td>
<td>string</td>
<td>Remote address (IP)</td>
</tr>
<tr>
<td>userAgent</td>
<td>string</td>
<td>User agent</td>
</tr>
<tr>
<td>time</td>
<td>time</td>
<td>Event timestamp</td>
</tr>
<tr>
<td>country</td>
<td>string</td>
<td>Country code (based on IP)</td>
</tr>
<tr>
<td>city</td>
<td>string</td>
<td>City name (based on IP)</td>
</tr>
</tbody>
</table>

Table 7.1: The schema associated with a topic.
7.2 Producer

A Producer publishes data onto the pipeline. Producers only care about topics. A producer does not need to know anything about any potential subscribers. Nor does a producer need to know anything about how the data they publish will be used. Producers only need to honor the schema associated with a topic and publish events as they occur.

As an example, the process of handling a user request is shown in Figure 7.2. The figure illustrates the process of handling a user login event. The event is initiated by the user as they click the log in button on a web page. First, the handler corresponding to the login route parses the request parameters which in this case is username and password. Next, the handler accesses the database to assert whether the user exists and whether the provided password is correct.

Simultaneously to the database lookup, meta-data from the request is extracted. This includes the remote address, user agent, time and IP-location. When the meta-data has been extracted and the database lookups have completed the final event message is constructed based on the outcome. In the second to last step the event is published to the analytics pipeline on a best-effort basis. This means that if the publish request fails to complete it will not be retried, it will however be logged as an error. Finally, the response is sent back to the user.

As seen in Figure 7.2, the event message is published before any response is returned to the user. This is because the details of the event depend on the outcome of the actual user login attempt. For example, if the user did exist but provided the wrong password the event message should reflect that. For other kinds of topics the event message might be ready to be published before the user request is completed. In such cases the event message can be published concurrently to the logic that handles the user request.

Producers integrate with the analytics pipeline using an API implemented
in a Go package. The API exposes a single function shown in Listing 2. The Publish function takes three arguments: a context for authentication, the HTTP request itself and a specific topic. Meta-data will be extracted from the HTTP request which will aggregate the topic message. The last argument is of variable length meaning that one or more of the same type of topic can be published. The function returns an error if the topic messages could not be published. The API uses a fire and forget best effort strategy when it comes to publishing messages. For example, errors are logged by the API but not returned to the sender.

```
// Publish publishes one or more topic messages.
func Publish(ctx context.Context, r *http.Request,
    topicMessages ...TopicMessage) error {
    // Check if all messages belong to the same topic.
    topic := topicMessages[0].Topic()
    for i := range topicMessages {
        if topicMessages[i].Topic() != topic {
            errMsg := "messages must belong to the same topic"
            return fmt.Errorf(errMsg)
        }
    }
    // For each message, set the base properties such as timestamp
    // and remote address. Then, base64 encode and add meta-data
    // to each message (in the newMessage function) before the
    // messages are sent over the wire.
    pubsubMessages := make([]*pubsub.Message, 0, len(topicMessages))
    for i := range topicMessages {
        topicMessages[i].SetBase(r)
        pubsubMessage, err := newMessage(topicMessages[i])
        if err != nil {
            return err
        }
        pubsubMessages = append(pubsubMessages, pubsubMessage)
    }
    _, err := pubsub.Publish(ctx, topic, pubsubMessages...)
    if err != nil {
        return err
    }
    return nil
}
```

Listing 2: The Publish function in the Go producer package. The listing shows an annotated version of the producer code.
### 7.3 Consumer

Consumers subscribe to topics in the analytics pipeline. Consumers maintain their subscription to a topic and either periodically polls the subscription for new data in batches or automatically receives new data. Consumers do not have any knowledge of the producers in the analytics pipeline. Like the producers the consumers communicate over HTTPS using JSON encoded data. There are two types of consumers: Long-term and Real-time.

Long-term consumers periodically poll a subscription for new data and store that data in BigQuery. The poll is initiated by a cron job which is a scheduled task. When the poll and insert is complete the application idles until the next phase. The scheduling can be tuned to change the rate at which the poll and insert is initiated. In each phase, up to 100 messages are pulled from a topic and inserted into BigQuery. The long-term consumer application is responsible for preparing the new messages for insertion into BigQuery and to acknowledge the messages so that the consumer will not receive the same messages again in the next phase. The work-flow for a long-term consumer is shown in Figure 7.3.

![Figure 7.3: Long-term consumer work-flow.](image)

For real-time consumers, messages are pushed to the HTTP endpoint specified in the subscription. Real-time consumers relays the messages from a topic to a web dashboard where every user get a continuous feed of new messages as they arrive. The real-time consumer application is responsible for maintaining the state of each live user and adding and removing users over time. The work-flow for a real-time consumer is shown in Figure 7.4.
7.4 Long-term metrics

By storing all the data for a topic we can query it at a later point and discover long term trends. Listing 3 shows a fairly simple query where the number of successful user logins are grouped by the location from which the login was requested from. By taking the result from the query and rendering it on a map we can get a visual answer to a basic question: Where are our users? This visualization is shown in Figure 7.5.

In this example the time range in the query is from the past month until current day as can be seen in the WHERE clause in Listing 3. This range could easily be altered to a narrower or much larger time window. For example, we could easily alter the query to only show the current day or year instead.

There are many types of interesting queries one can ask once we have the data. From a security standpoint we could further explore the capabilities of the user login topic to mitigate potential security issues. For example, we can show the number of successful and failed logins for each day. The query is shown in Listing 4. Using this data one can easily identify if there has been an unusually large number of failed login attempts. This could be an indication that someone is attempting to guess a user password or that the login form needs a re-design.
Figure 7.5: Where are our users? The map shows user logins grouped by the location of the requests. Map data by Google.

```
SELECT COUNT(*) AS Logins, Country, City
FROM userlogin
WHERE Exist = true
    AND CorrectPassword = true
    AND Disabled = false
    AND Time > DATE_ADD(CURRENT_TIMESTAMP(), -1, "MONTH")
GROUP BY Country, City
ORDER BY Logins DESC
```

Listing 3: User logins grouped by geographical location in BigQuery.
SELECT t1.year AS year, t1.month AS month, t1.day AS day, t1.hour AS hour,
    IFNULL(INTEGER(t1.logins), 0) AS logins,
    IFNULL(INTEGER(t2.wrongPassword), 0) AS wrongPassword
FROM (SELECT
    YEAR(time) as year, MONTH(time) as month, DAY(time) as day,
    HOUR(time) as hour, COUNT(*) AS logins
    FROM userlogin WHERE exist = true AND correctPassword = true
    GROUP BY year, month, day, hour
    ORDER BY year DESC, month DESC, day DESC, hour DESC) AS t1
LEFT OUTER JOIN
    (SELECT
        YEAR(time) as year, MONTH(time) as month, DAY(time) as day,
        HOUR(time) as hour, COUNT(*) AS wrongPassword
        FROM userlogin WHERE exist = true AND correctPassword = false
        GROUP BY year, month, day, hour
        ORDER BY year DESC, month DESC, day DESC, hour DESC) AS t2
ON t1.year = t2.year AND t1.month = t2.month
    AND t1.day = t2.day AND t1.hour = t2.hour

Listing 4: User logins attempts for each day in BigQuery.
7.5 Real-time metrics

Figure 7.6 shows an example of real-time data displayed on a web-based dashboard. New messages are relayed to a web application as they arrive at the consumer server depicted in the bottom right corner of Figure 7.1. In this example, data comes from a topic that describes client log records. The log record topic schema contains a status code field which is shown on the y-axis of the chart. The chart updates continuously as new records come in.

In this particular example, status codes in the 20000 range correspond to a successful action. Status codes in the 30000 range correspond to an exception or error. Real-time metrics make it possible to make informed decisions regarding whether the clients are experiencing issues right now.

![Client logs real-time view](image)

Figure 7.6: Client logs real-time view.
7.6 Management tools

Management is the last piece of the analytics pipeline and consists of a web interface for administrators. The management tools provides useful debugging information such as latency statistics for real-time messages and logs for batch jobs. The tools are primarily used to manage topics (Figure 7.7) and subscriptions (Figure 7.8) for producers and consumers via Cloud Pub/Sub. BigQuery datasets and tables can also be managed directly from the web interface as seen in Figure 7.9.

Figure 7.7: Topic management.
Subscribers

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Topic</th>
<th>AKL deadline (s)</th>
<th>Push endpoint</th>
</tr>
</thead>
</table>
| 1  | clientlog | clientlog | 60               | https://host/live/project/clientlog/clientlog-
|    |           |        |                  | real-time              |
| 2  | clientlog | clientlog | 600              | Unsubscribe            |

New subscription

Name
my-sub

AKL deadline (s)
128

Push endpoint

Create

Figure 7.8: Subscriptions management.

Tables

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clientlog</td>
</tr>
<tr>
<td>2</td>
<td>ping</td>
</tr>
<tr>
<td>3</td>
<td>userlogin</td>
</tr>
</tbody>
</table>

New table

clientinfo

Select a schema

Shows information about client setups.

Create

Figure 7.9: Table management.
Chapter 8

Results

This chapter introduces the results of the implementation and architecture with respect to the evaluation criteria outlined in Chapter 5.

8.1 Flexibility

Integration with the analytics pipeline should be straight-forward and simple

The process of integrating a new or existing code base with the analytics pipeline involves only a few steps. The only management set-up required before a producer can start sending messages to a topic is the creation of the topic itself. Because the producers are not coupled with the consumers in any way it is not necessary to set up the complete pipeline before the producer starts sending messages. From a producer standpoint, the integration is complete once the topic schema is decided and the topic created.

Because of the way a topic schema has a corresponding table in BigQuery, the work for a long-term consumer whose job it is to ingest data for long-term storage amounts to little more than just relaying data from a subscription and inserting it into BigQuery. The consumer only needs to know from what subscription it should pull and which table it should insert data into. In terms of management, the consumer needs to set-up a subscription prior to starting. For long-term consumers this means simply creating the subscription and naming it. Real-time consumers need to specify an HTTP endpoint that new messages will be pushed to.

The API exposed to the developer should have a minimal footprint when used in the code

For producers, integration with the analytics pipeline is achieved by using the client library Go package developed for this project. The public
part of the client library exposes only a single function: `Publish(context, request, topicMessage)`. From the developers perspective, the only variable part of the input to `Publish` is the topic as the other inputs remain the same. This is achieved by abstracting the task of authentication, validation and serialization from the developer whose task it is to integrate with the analytics pipeline. The code footprint of integrating the client library depends on the complexity of the event to capture. If the schema of a particular topic only consists of a few properties the integration of a producer into an existing code base could consists of only a single line of code.

The analytics pipeline should be platform and language agnostic

All communication in the analytics pipeline is done over HTTP using TLS (HTTPS) by default. The data-interchange format for all forms of communication is JSON. Both HTTP and JSON are ubiquitous technologies that are present in virtually every platform. This means that direct integration to the analytics pipeline can be achieved from practically any type of application.

It should be easy to manage topics, producers and consumers

Topics, and consumers are managed from a web-based management interface. The management interface provides the basic actions that a user needs to create new topics and manage subscriptions to that topic. New tables can be created based on the predefined schema of topics. The management interface also acts as a means of access control. For example, stale consumers can be removed by unsubscribing them from a topic. For persisted storage, datasets and tables can be deleted and recreated.

The data format should not impose any unnecessary restrictions on which types of events can be described

The data-interchange format used is JSON which provides flexible attribute-value data objects that can be nested. In terms of available data-types the lowest common denominator are the data-types available to BigQuery: string, integer, float, boolean, time and records which is a nested collection of the other available types. In effect, these types are the only ones that can be used as all topics schemas must be compatible with BigQuery.

Even though the underlying technologies have the capability to use nested records it was decided that topic schemas must have a flat structure. The reason for this restrictions is twofold. It simplified the management tools for BigQuery tables significantly as the code that generated BigQuery tables from Go structs could be kept simple. Furthermore, a flat structure is more in line with how one typically normalizes tables in a RDBMS.
It should be easy to ask questions and get answers from the data that is persisted

Persisted data is stored in BigQuery which is a structured database with an SQL-like query language. SQL is a powerful and well understood language for data queries. In BigQuery, every column is indexed. This means that every property of a topic schema can be queried with equal level of flexibility without any additional work required.

### 8.2 Scalability

#### The analytics pipeline should be able to scale as the load increases

The analytics pipeline is built on top of the services of the Google Cloud Platform and as a result the implementation can only be as scalable as the underlying platform. In this project App Engine was used for both producers and consumers. As the number of requests increases to an App Engine application, more instances start-up to serve the application in order to keep up with the demand. In the context of cloud computing this is referred to as automatic scaling. This makes it easy to scale out by adding more nodes both on the producer and consumer side.

The scalability of BigQuery and Cloud Pub/Sub is difficult to assess as they are both provided as services with very few configurable options.

#### Topics and subscriptions should remain maintainable as more are added

Topics and subscriptions are uniquely identify by their name. This makes it easy to create humanly readable and easily identified topics and subscriptions. As the number of topics grow the maintenance over-head is mainly in keeping the schema definition in sync with the code base. For example, if a topic schema definition changes the code that uses that particular topic needs to be updated as well. The same goes for the corresponding table in BigQuery. Because of the way BigQuery works, updates to tables are not possible. However, a new table can be created with the updated schema at which point data can be moved to the new table.

#### Real-time data should have sub-second latency between servers within the analytics pipeline

The 95th percentile latency for real-time consumers between different servers on Google Cloud was 843 ms. In other words, with the exception of a few outliers 95% of all messages had sub-second latency. This was measured by comparing sent and receive timestamps for 7800 userlogin topic messages
collected with nanoseconds precision over a period of 6 days. The duration was calculated by comparing the timestamp set by the producer before JSON serialization and base64 encoding with the timestamp set by the consumer after base64 decoding and JSON deserialization. The mean latency was 126 ms with a standard deviation of 484 ms. The latency distribution is shown in Figure 8.1.

![Message latency in milliseconds](image)

Figure 8.1: Latency in milliseconds for each message. The y-axis is in logarithmic scale. The dashed horizontal line shows the 95th percentile.

**Long-term data should remain easily accessible and workable even when million of records have been stored**

As described in Section 6.1, BigQuery has a relative high latency involved with both inserting and querying data. The advantage of BigQuery is that as more data is added the latency does not get noticeably worse. Data can also be exported to a csv file for further work and data analysis that is not easily done within BigQuery.

### 8.3 Robustness

**A system or network failure should not affect the system as a whole**

The decoupled design of the analytics pipeline is such that any part of the system could fail without causing ripple effect. Since producers and
consumers never talk directly to each other a failure on either end will not affect the other. Messages are replicated and persisted to Cloud Pub/Sub so even if a consumer is lagging behind on its subscription due to an outage, the consumer can catch-up once it starts working again.

**Loss of service availability should be handled gracefully and not affect the rest of the system**

If a long-term consumer is unable to pull for new messages because the service is not available, it will simply try again during its next cycle. For producers, unsent messages are logged but not saved. This is in the spirit of keeping the system simple and maintaining a best-effort strategy. It would be necessary to further develop the client library to be able to cache unsent messages. Finally, since all communication is done over HTTP, which is a stateless protocol, a message is either received or it is not.

**Erroneous data should be explicitly handled and logged**

The topic schema only specifies what data-type should be associated with a property, not what values it can have. It is up to the developer to ensure that data sent from the producers handles erroneous data according to application requirements.

**Event messages should be delivered exactly once**

Cloud Pub/Sub does not guarantee that a message is delivered to a subscriber exactly once. If multiple subscribers, and by extension, consumers, use the same topic there is no guarantee that any two consumer will not receive the same message. Consumers need to acknowledge the fact that they have received a message otherwise they can not receive another one.

To help consumers identify duplicate messages a unique ID is inserted into each message before it leaves the producer. This ID is used by long-term consumers in order to discard duplicate message. This prevents long-term consumers from inserting the same message into BigQuery more than once. For real-time consumers there is no such handling and users may occasionally receive duplicate messages.
Chapter 9

Discussion & conclusion

9.1 Decoupled design

The message oriented communication of publish/subscribe lends itself well to analytics data. Typically, such data does not need the same kinds of consistency and reliability guarantees as the core business data. The publish/subscribe pattern also implies late binding between components. As an example, a developer whose responsibility is to integrate a new producer into the pipeline needs no knowledge of any possible consumers. As far as the producer is concerned, the number of consumers is irrelevant and transparent to its operation. In the same way, when someone wishes to get to the analytics data by setting up consumers there is no need to interfere with the producers. The data is available, you just need to setup the consumption. Software is extremely malleable. It changes fast. The concept of a topic schema plays an important part here because it represents the contract between producers and consumers. A topic schema remains unchanged even if the rest of the system changes. This decoupling is profound and enables decomposition of large programs into smaller independent services. This means that users can get to the data they want, when they want it. Without either having to decide upfront who should have what data or having to change everything once someone says I want that data!

9.2 Familiar query language

As shown in Chapter 3, MapReduce technology is commonly an integral part of many data-analytics systems. However, for the purposes of this master thesis, BigQuery was used as means to store and analyze data. Performance differences and use-cases aside, one of the common criticisms of MapReduce is the inherent complexity in writing MapReduce jobs. A query language such as SQL that can be used interactively makes it easier to perform exploratory ad-hoc data-analysis. SQL is a mature language with a
large community of users. The query language is well understood among database professionals, data scientists and developers. But there is also a wealth of other users, such as analysts, customer support professionals and others that are familiar with using SQL as a means to retrieve and analyze information. SQL is the *lingua franca* of data analysis. The familiarity of the SQL-like language found in BigQuery brings big data analysis to the masses.

### 9.3 Tools and plumbing

Lin et al. argues that the challenges of connecting different homogeneous software components together, which they refer to as the *plumbing*, is often left out of research. They argue further that often considerable effort is spent on making sure things work together before one can even begin to extract insights from the data [18]. Organizations who are interested in the ability to gain insights from data that moves fast would benefit from knowing about these costs upfront. This is one of the reasons that the development of the management tools was crucial. By developing tools we can better understand how the software components work together. Arguably, the development and maintenance of such tools are necessary to the success of any data analytics pipeline.

### 9.4 Cloud-based architecture

A major concern in the development of a system that consists of many different components spread over many different machines is provisioning of infrastructure. This entails the actual machines, networking and power equipment as well as other components on which everything runs. For the purpose of this project however, those details could largely be ignored. This is because the implementation makes extensive use of both platform and software provided as services meaning that the operation and maintenance to a large extent is transparent to the user. As previously mentioned, the loosely coupled design of the architecture has many benefits but it also means that the complexity of the infrastructure increases. This complexity still exists, but the responsibility has shifted from the users to the service providers. One of the most important and identifiable aspects of cloud-based computing is the scale out nature meaning that you typically pay for what you use, thus avoiding large upfront costs. This flexibility and low-barrier to entry of cloud-based services enables exploration and makes it viable for small organizations and institutions to develop the kinds of system that previously only large organizations would have the resources for.
9.5 Future work

The management tools outlined in Section 7.6 were designed to provide the basic functionality required to facilitate experimenting during development. They also proved to provide sufficient enough control needed to maintain the system once the implementation had reached a more stable phase. In order to improve the scalability, and by extension, the usefulness of the system, it is imperative that the system provides a higher degree of configuration automation.

As presented in Chapter 7, the system defines two types of consumers: long-term and real-time. Because of this it would be useful to have the ability to create a new consumer and define it as either of the two types. This action would then automatically create the necessary datasets, tables and views based on the requirements.

Topics are central to the ideas of this project. In the current implementation the schema of the topics are defined in a source code package that is accessible to the application that uses them. Go is a statically-typed and compiled language. This means that applications need to be rebuilt in order to use a newly defined topic schema. One possibility is to instead provide a topic service application which runs independently and provides an REST API through which other applications can then create new topics, ask for schema details etc.

As mentioned in Section 6.2, Cloud Pub/Sub does not guarantee that messages will be delivered in the order that they were sent in, neither does it guarantee that no messages will be lost. Because of this it was necessary to validate the ratio of messages sent and received during testing. These tests were done in an ad-hoc manner and should be replaced with functionality that continuously does random sampling of message delivery across different topics and subscriptions to make sure that the services are working as intended under acceptable conditions.

As outlined in Chapter 5, the aim of this project was to engineer a solution for a cloud-based analytics pipeline by identifying the components and infrastructure necessary. With the infrastructure in place to facilitate the collection of data for both long-term and real-time use we can now easily start to add more interesting sources of data, define more topics and use it to our advantage in our decision making. Finally, one of the most interesting ideas in the context of big data analytics is the prospect of combining long-term historic data together with real-time live data. Fan and Bifet outline the future challenges for big data analytics and argue that the architecture for such a system is not yet clear [9].
List of Figures

2.1 A web page login form. .................................................. 9

7.1 Architecture of the data analytics pipeline. ...................... 22
7.2 Producer work-flow for the user login topic. .................... 24
7.3 Long-term consumer work-flow. ................................. 26
7.4 Real-time consumer work-flow. ................................. 27
7.5 Where are our users? ............................................ 28
7.6 Client logs real-time view. ....................................... 30
7.7 Topic management. ............................................... 31
7.8 Subscriptions management. ..................................... 32
7.9 Table management. ............................................... 32

8.1 Latency in milliseconds for each message. ................. 36
List of Tables

7.1  The schema associated with a topic. . . . . . . . . . . . . . . 23
List of listings

1. A concurrent Go program ........................................... 21
2. The Publish function in the Go producer package ............ 25
3. User logins grouped by geographical location in BigQuery ... 28
4. User logins attempts for each day in BigQuery ................. 29
Bibliography


45


