CROSS City Cloud:
Extension, Deployment and Operation of a Smart Tourism Application relying on Location Certificates

Lucas de Haan Vicente

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Filipe Leitão Pardal
Dr. Samih Eisa Suliman Abdalla

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. Luís Manuel Antunes Veiga

October 2022

I dedicate this work to my beloved, late grandfather Joop de Haan. You were a major inspiration for me while growing up, and I would not be the person I am today without the impact you had in me.

Acknowledgments

I am extremely grateful to have had the opportunity to work with my advisor Miguel Pardal and co-advisor Samih Eisa. Both their advice and collaboration were critical from the ideation all the way up to the delivery of this work. I was fortunate enough to be able to not only learn from all the active and past contributors of the SureThing project, but also add my own contribution to it.

I would want to offer my deepest gratitude to Prof. Nuno Santos and his team for their cloud support and funding; without their assistance, the deployment of our solution would not have been conceivable. I would also like to express my heartfelt appreciation to Rui Claro for his remarkable endeavor in gathering Wi-Fi data around Lisbon. His commitment was vital to the evaluation of the developed solution.

I would also want to thank my friends and colleagues Rafael Figueiredo and Ricardo Grade for all of the ideas and discussions we had throughout our journey at the SureThing project. I would like to thank in particular my friend André Augusto, with whom I had the opportunity to collaborate on several projects throughout our academic pursuit and who was always available to encourage me to accomplish this work.

Last, but certainly not least, I would like to thank my father Nuno, my mother Renate and my sister Inês for their unconditional support throughout this endeavor and my academic years. They were able to provide me with a perfect stable environment in which to fulfil this work, which benefited positively to my mental health. I am incredibly grateful to my parents for providing me with the opportunity to pursue a higher education in a field that I am passionate about.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and through project with reference PTDC/CCI-COM/31440/2017 (SureThing).

Resumo

Lisboa é uma das cidades mais visitadas do mundo, recebendo milhões de turistas por ano. Muitos destes visitantes usam aplicações móveis para descobrir pontos de interesse na cidade, recorrendo ao posicionamento geográfico. Apesar desta forte dependência, a maioria das aplicações está suscetível a ataques de falsificação de localização.

O *CROSS City* é uma aplicação móvel desenvolvida para o turismo inteligente, que recompensa utilizadores por completarem circuitos turísticos, emitindo certificados de localização, que oferecem proteção contra falsificações. No entanto, o *CROSS* exige o uso de infraestrutura específica para validar o período de visita. Para além disso, sendo um protótipo, os operadores do sistema têm dificuldade em garantir a sua confiabilidade, segurança, manutenibilidade e observabilidade.

Neste trabalho propomos o *CROSS City Cloud*, um sistema desenhado de raiz com computação em nuvem para a certificação de localização numa aplicação móvel, capaz de produzir e validar provas de localização com granularidade temporal usando a infraestrutura pública de redes *Wi-Fi*. Foram desenvolvidos componentes para a computação das redes estáveis e voláteis de um determinado ponto de interesse. Os novos componentes foram instalados na Nuvem da *Google*, incluindo um plano de controlo que facilita a sua operação. O caso de uso de turismo inteligente demonstrou a viabilidade de uma plataforma de Certificação-de-Localização-como-um-Serviço em ambientes de computação em nuvem. A avaliação experimental testou o desempenho e escalabilidade de cada componente em cenários reais. Foram alcançadas taxas de sucesso de 61.11% e 63.89% na determinação da localização e intervalo temporal de visitas turísticas, respetivamente, validando a solução.

Palavras-chave: Prevenção de Falsificação de Localização, Prova de Localização, Adaptação ao Contexto, Segurança Móvel, Internet das Coisas, Computação em Nuvem

Abstract

Lisbon is one of the world's most visited cities, attracting millions of tourists each year and many of them use smartphone apps to discover points of interest. Although these apps heavily rely on geographic positioning information, most of them are susceptible to location spoofing. CROSS City is a smart tourism application that uses location certification to reward users for completing tourist itineraries. However, CROSS requires the use of purpose-built infrastructure to validate the visiting period of a location claim. Furthermore, since it is a prototype, it is difficult and costly for system operators to ensure its reliability, security, maintainability, and observability.

In this work, we introduce CROSS City Cloud, a cloud-native location certification system for mobile applications, capable of producing and validating time-bound location proofs using the publicly available Wi-Fi network infrastructure. We present an architectural extension composed of the components necessary to integrate scavenged Wi-Fi network observations and compute the stable and volatile networks of a given location. We designed and deployed a cloud solution of the system, including an additional control plane to ease service operation, to Google Cloud Platform. The smart tourism application use case was utilized as a testbed, and demonstrated the feasibility of a Location-Certification-as-a-Service platform embedded in cloud computing environments.

Our evaluation stresses each component of the system in various aspects, such as performance and scalability, including real-world scenario assessments. The achieved stable and volatile set match success rates of 61.11% and 63.89%, respectively, validated our solution for the expected use case.

Keywords: Location Spoofing Prevention, Location Proof, Context-Awareness, Security, Internet of Things, Cloud Deployment.

Contents

List of Tables

List of Figures

Nomenclature

BSON	Binary JavaScript Object Notation
JSON	JavaScript Object Notation
LBS	Location-Based Service
SQL	Structured Query Language
XML	Extensible Markup Language
SSID	Service Set Identifier
FaaS	Function-as-a-Service
IaC	Infrastructure as Code
LXC	Linux Container
NIST	The National Institute of Standards and Technology
PaaS	Platform-as-a-Service
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
REST	Representational State Transfer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1

Introduction

Modern mobile applications and services rely heavily on geographic location to provide users with relevant context-aware and appropriate information. Practical use case services include: *map navigation*, *smart tourism*, *weather services* and *location-based games.* Several techniques can be used to provide applications with contextual location information. However, many of these employ techniques which do not thoroughly verify the location information they consume, making the applications and services vulnerable to various location spoofing techniques [LB10]. Once we have vulnerabilities on valuable applications and services, it is only a matter of time until malicious actors attempt to exploit them.

In order to combat and provide protection against the aforementioned attacks, location certification or proof systems [ZC11, CCCDP13, WPZM16] provide a means for producing reliable digital certificates (digitally signed proofs) attesting an individual's presence at a claimed geographical location. The generated certificates can subsequently be utilized by an Location-Based Service (LBS) to validate location claims. Location certification systems are typically capable of creating proof for location related information, such as a single specific location or a full travel route history, but often fail to take into account the time of occurrence without the usage of auxiliary trusted infrastructure at each point of interest.

An initial version (v1) of CROSS City [MCP20] (*loCation pROof techniqueS for consumer mobile applicationS*) was developed for the smart tourism use case, in which tourists interact with existing infrastructure at points of interest in the city using their mobile devices. Tourists are rewarded for engaging in this interaction, which in turn motivates them to continue using the application. However, such rewards also entice bad actors to illegitimately attempt to obtain them. To combat this, the interaction must produce verifiable information about a tourist's location that the LBS can use for validation. While the initial version of CROSS offered multiple strategies for producing location proofs, such as the Scavenging, TOTP (Time-based One-time password), and Kiosk. None of these strategies attested for the time of visit without augmenting each point of interest with additional trusted infrastructure.

To eliminate the need for additional infrastructure, further improvements can be performed on top of the *network scavenging* location proof strategy [CEP22]. These improvements rely on the data collected from the tourist's smartphone sensors and its comparison to previously obtained Wi-Fi data from the claimed location, to proof visits with temporal granularity. The CROSS City v1 architecture lacks the necessary components and protocols to properly support these enhancements.

Furthermore, the initial version of CROSS was a prototype not designed with production software system properties in mind, such as reliability, availability, scalability and performance. It also lacks a systematic and automatic approach for configuring and deploying its various components. CROSS v1 also does not provide operators with the information to assess the system health status of its back-end services or a way to determine the cause of failures. The aforementioned problems are significant challenges towards providing a Location-Certification-as-a-Service platform to system operators managing location certification dependent applications.

In this work, we present *CROSS City Cloud*, a cloud-native location certification system with support for time-bound location proofs, serving as a testbed framework, to demonstrate the feasibility of embedding a location certification framework into public cloud computing technology to provide a Location-Certification-as-a-Service platform. This work was developed within the SureThing framework [FP18]. This framework provides data formats and utility libraries which facilitate the creation and validation of location certificates for Internet-connected devices. Although, CROSS is based on the SureThing framework, the v1 prototype lacks the usage of its data types and libraries, in the smart tourism use case

1.1 Contributions

This work makes four distinct contributions:

1. We extended the server-side architecture of CROSS with an additional data management layer and components to properly support location proofs with temporal granularity;

2. We selected and implemented the appropriate service abstractions and cloud offerings for each architectural component, based on the use case requirements and the predicted workloads;

3. We delivered CROSS services on-demand by leveraging cloud computing, involving the deployment and operation of our system on Google Cloud Platform using Docker and

Kubernetes;

4. We evaluated each resulting architectural layer in different aspects involving location and time-bound proof feasiblity, scalability, and performance from both a system and a user perspective. Also, alternative implementation options were assessed.

1.2 Dissertation Outline

The remaining chapters of the document are organized as follows. In Chapter 2, we start with a detailed presentation of the CROSS City prototype. Next, we go over a data analysis model aiming to ensure time-bound location proofs, built on top of its scavenging strategy. In addition, a survey and analysis of data storage and processing components, as well as relevant architectures to the CROSS City Cloud extension, are presented. Furthermore, modern cloud computing technology is discussed, as to support the deployment and operation of the system components. Chapter 3 specifies the solution in depth, including the architecture extensions, the cloud deployment and the service operation. Chapter 4 presents the evaluation performed to assess the solution and its various layers, in aspects such as performance and scalability. Chapter 5 summarizes the conclusions and presents future work.

In Appendix A a detailed database type analysis is conducted. Finally, in Appendix B, we analyse and select the proper methodology for automating the system configuration and deployment, as well as discuss the tools to secure the environment at the cloud provider level.

Chapter 2

Background and Related Work

In this Chapter, we present concepts and technologies that are relevant for our work.

In Section 2.1, we start by presenting the initial CROSS City prototype [MCP20], analyze its shortcomings, as well as a related work [CEP22] aimed at addressing those limitations. In Section 2.2, we survey the state of the art in database models and their respective implementations. In Section 2.3, we study the distinct methods of data processing, and in Section 2.4 we discuss two data processing architectures which incorporate data processing and storage components. In Section 2.5, we discuss the current cloud computing landscape, including the virtualization technologies present at its core, delivery and deployment models, and the services currently supplied by the major providers. We conclude this Chapter with Section 2.6 where we summarize the literature review.

2.1 CROSS City Application

Maia et al. [MCP20] proposed an initial prototype version of CROSS (v1), a system that implements a set of location proof techniques for consumer mobile applications. In CROSS, tourists are rewarded for completing itineraries of points of interest (PoI) in the city; The mobile device of each tourist interacts with Wi-Fi and other existent infrastructure and records verifiable information (*trip logs*), for later validation of location claims.

2.1.1 Architecture

CROSS v1 uses a client-server model consisting of a mobile application and a centralized server with a database component, as illustrated in Figure 2.1. The server is responsible for handling the validation of location evidence submitted by the tourists. The server contains a persistent module with domain data such as user information, points of interest, tourism routes, estimated rewards and the set of APs (identified by their SSIDs) expected to be present at each location.

Regarding the technologies used for implementation, the client prototype is running Android 4.4 version or higher. Communication between the client and the server is done through HTTPS. The server is implemented in GO and exposes a REST API with JSON payloads. The persistent module is a PostgreSQL relational database, which stores all the domain data.

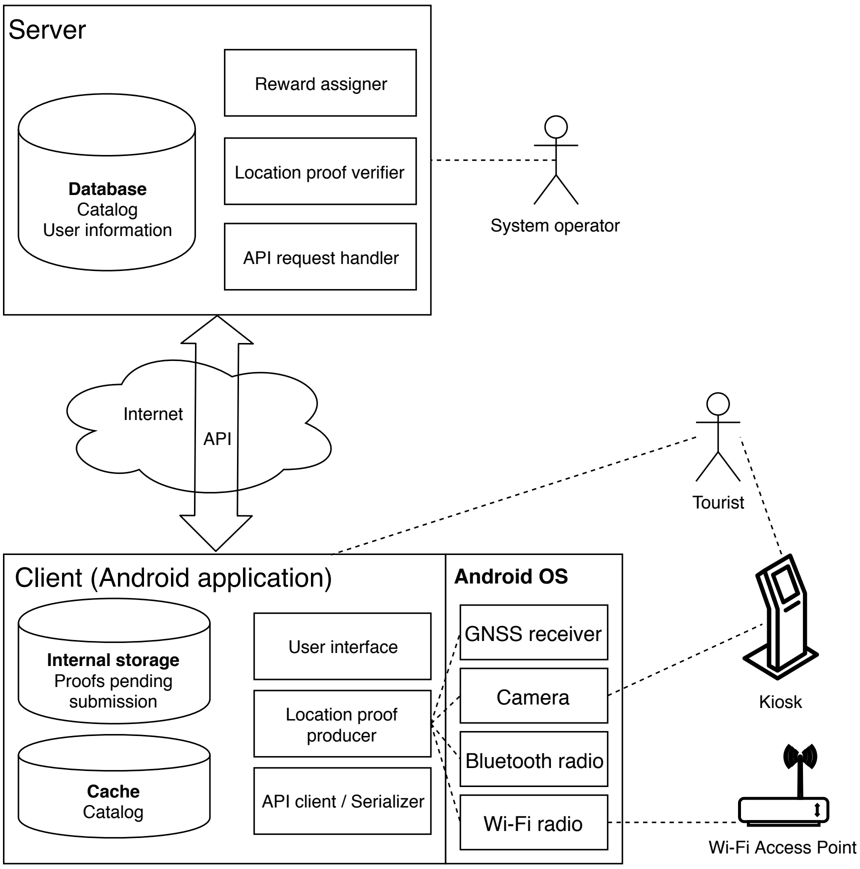

Figure 2.1: Overview of CROSS v1 architecture [MCP20].

2.1.2 Location Proof Techniques

CROSS v1 is able to employ three distinct strategies for location verification. Each strategy provides different levels of security by trading off infrastructure complexity and operational costs:

- ***Scavenging strategy***: Users collect Wi-Fi traces with associated timestamps at the point of interest and store them as evidence. The stored data is compared against the list of known networks at that location and are accepted based on the amount of matches. This strategy has a reduced setup cost, however it provides a weaker level of confidence, since an attacker could forge a trip log after knowing the list of public networks;

- ***TOTP strategy***: Leverages a Time-based One-time Password similar to the proposed in RFC 6238 [MMPR11], in the broadcast SSID. This strategy requires the deployment of a customized Wi-Fi AP that is dynamically changing the broadcast SSID in a set period. Only the Wi-Fi AP and the CROSS server share a secret which is used to produce and validate the codes. This strategy is able to attest for both a user's location and visiting period, at the expense of setup cost and further synchronization between the custom AP and the server. From the point of view the client, this strategy is exactly the same as the Scavenging strategy;

- ***Kiosk strategy***: Clients produce location proofs by interacting with a trusted kiosk device, which will sign the required information to be later verified by the server. Similarly to the TOTP strategy, kiosks are required to be synchronized with the server. This strategy is more inconvenient for the clients and requires extra infrastructure, the kiosks themselves, however it does provide more assurance.

2.1.3 Time-Bound Location Proofs Based On Scavenging

The ability to offer verifiable information with regards to a location time of visit strengthens the dependability of the location proofs generated, however when using the TOTP strategy each point of interest needs to be augmented with a customized Wi-Fi AP to dynamically broadcast its SSID value synchronized with the server.

Claro et al. [CEP22] collected a dataset of Lisbon *hotspots*, developed a *data model* and *algorithms* to determine the location and time interval of a tourist visit, as precisely as possible with the existing data. Three distinct entities are defined, *prover* (the user of the system trying to prove his presence at a location), *witness* (neighboring user of the prover that provide the system with location proof attesting for the prover), and *verifier* (entity that validates the location proof submitted by the prover). Their approach leverages diverse ad-hoc witnesses to observe both *long-lived* and *short-lived hotspots* to detect the location and prove the time of visit, respectively, of other users. A prover's - the user of the system trying to prove his presence at a location - location claim uses as evidence a collected set of Wi-Fi Access Point SSIDs, referred to in the model as *observations*. Three time windows are defined in the model to bound the observations for verification:

- *Epoch*: The most encompassing time frame. Only observations collected within the defined epoch time window will be used to compute the stability of the Wi-Fi networks at each location. These networks do not change and, as such, provide location verification. This process is executed at the start of the system;

- *Period*: A subdivision of an *epoch.* Only observations collected within the defined *period* time window will be used to compute the volatility of the Wi-Fi networks at a given location. These networks change and, as such, provide a time bound for simultaneous observation;

- *Span*: A subdivision of a *period.* A *span* is the interval formed by the time of visit in the location claim (t_p) and an additional parameter *delta* (δ) between $t_p - \delta$ and $t_p + \delta$. The ideal *span* is the smallest window, and consequently the smallest *delta*, where evidence was found to verify a location and time claim. Prover and witness must share observations in the same span.

As mentioned previously, the verification of a prover's time of visit depends on the co-location with witnesses. This involves the identification of the volatile set of networks at that specific location from matching observations collected by co-located witnesses, within the defined *period*, and by the prover. The smallest interval (*span*) where evidence is found to verify the time claim provides the highest accuracy.

This approach does not require any sort of communication between witness and prover, it only requires periodic submission of sensed Wi-Fi traces to the server that acts as *verifier.*

2.2 Data Storage

CROSS must store domain and user collected data, such as Wi-Fi and other signal observations, in both its raw and processed forms. Therefore, it is important to determine the appropriate database types for integrating each data. To accomplish this, we surveyed existing data models and their mainstream implementations.

The data model of a database does not by itself translate directly into any sort of enhancement, for example, *MongoDB* is not simply *better* than *PostgreSQL* due to its employment of the document-oriented data model, instead of relational. The distinction between database systems is based on the unique implementation of a given data model via its components, as well as the assurances that each one may provide. Section 2.2.1, introduces some key concepts. Sections 2.2.2 to 2.2.5 will be structured with a general definition of the data model and an analysis of a specific implementation. Section 2.2.6 summarizes the ideas discussed throughout the database analysis. In Appendix A, we discuss in more detail.

2.2.1 Key Concepts

Before delving into the database type analysis, we introduce the database concepts [SKS$^+$02] necessary to grasp, in order to understand potential benefits and drawbacks of the distinct properties and guarantees supported by each database.

Transaction, A group of commands that must behave as a single atomic command, meaning they must all succeed or fail as a single unit, and their consequences, depending on the isolation level, should not be visible to other sessions until the transaction is completed.

Atomicity, The property of a transaction in which all or none of its operations complete as a single unit. Furthermore, if a system failure happens while a transaction is being executed, no partial results are visible following a recovery.

Consistency, The property of ensuring that data in a database is always in accordance with integrity constraints. Transactions may transiently break some of the constraints before committing, but if such violations are not rectified by the time the transaction commits, the transaction is immediately rolled back.

Isolation, The property that ensures that the effects of a transaction are not visible to concurrent transactions before it commits.

Durability, The guarantee that after a transaction has been committed, the changes will persist even if the system crashes.

ACID Transaction, A transaction is deemed to follow ACID semantics if it ensures the properties of *Atomicity*, *Consistency*, *Isolation*, and *Durability*, as described above. These properties are meant to ensure validity in concurrent operation, and even in the case of errors or power outages.

Scalability, Capability of the database to handle a growing amount of load with the potential to be enlarged in order to accommodate that growth. Scaling can be done either horizontally or vertically. Horizontal scaling implies adding or removing database nodes, while vertical scaling means adding or removing resources to a single node [Kle17].

According to the CAP theorem [Bre12], a distributed database may have, at most, two of three properties:

- **Consistency** (C): Database node maintains a single up-to-date copy of the data. Every read request visualizes the most recently modified dataset or does not return;

- **Availability** (A): Every request receives a response, i.e. the system is always available to respond to both read and write requests;

- **Partition Tolerance** (P): Tolerance to network partitions, even if some communications are lost or delayed, the system continues to function.

It is crucial to emphasize that, as mentioned by Brewer [Bre12], partitions are commonly viewed as unavoidable in distributed systems across a "wide area". For this reason, most practical distributed systems must be designed to yield either consistency or availability, becoming AP or CP, respectively. It is also worth noting that the system is only truly forced to make this sacrifice during a partition.

2.2.2 Relational

As initially proposed by Codd [Cod02, Cod89], a *relational database* is a collection of related data structure types, in the form of tables. Tables are organized into columns and rows, each column represents an attribute, storing a single type of data. Each row of a table holds an entire record or tuple, uniquely identified by a key. Data retrieval, manipulation or definition, in most relational databases, is achieved through the execution of Structured Query Language (*SQL*) statements. This manipulation consists of a series of relational algebra operators, which when applied to a certain relation (table) produce a new relation, causing a transformation.

PostgreSQL[1] is an open source object-relational database management system (ORDBMS) implemented in C, based on POSTGRES [SR86, SRH90]. PostgreSQL utilizes the standard Structured Query Language (*SQL*) which includes statements for defining and manipulating data. It supports ACID semantics for transactions. PostgreSQL's replication method is *Primary-Backup* where a cluster is formed of a single primary node, responsible for receiving data modifications and forwarding them to the backup nodes. Replication can be done either synchronously or asynchronously, however by default it is done synchronously to ensure strong consistency. PostgreSQL does not provide a horizontal scalability solution such as sharding, instead partitioning is done solely at the table level, to improve query performance, using one of three methods: Range, List, Hash. It does not provide a storage engine capable of running in-memory only.

2.2.3 Key-value

As described by Han et al. [HHLD11] and Seeger [See09], the *key-value data model* consists of a "key-value" relationship, where the value, representing the actual stored data, is indexed by a uniquely identifiable key. These keys are typically strings and the data is generally a programming language primitive, facilitating the marshalling between the key-value database.

Redis[2] is an open source in-memory data structure store written in C, which maps keys to

[1] https://www.postgresql.org/

[2] https://redis.io/

five different types of values. Redis trades-off a traditional and robust query language to have a simplified schema-less key-value data model. Redis instances are able to execute server-side code written in Lua. Redis can only support transactions with no rollback mechanism, thus not ACID compliant. Redis supports a primary-backup replication scheme, in which the replicas are copies of a single master instance, however it lacks automated failover in the event of a master's failure. Redis provides asynchronous replication trading a stronger level of consistency for higher availability. Redis offers a *Hash Slot* method for sharding data from a dataset among several nodes. By default, operations are done over a dataset present only in-memory, however data may be persisted by regularly dumping the dataset to disk.

2.2.4 Column-oriented

As outlined by Han et al. [HHLD11] and Abadi et al. [ABH09], *column-oriented databases* use a table as the data model, however each database table column is stored separately with attribute values belonging to the same column stored contiguously, compressed, and densely packed. This contrasts with traditional relational database systems that store entire records (rows) one after the other and associate tables. Reading a subset of a table's columns becomes faster, potentially at the expense of excessive disk-head seeking from column to column for dispersed operations. Each column is typically handled by a separate process, allowing for concurrent process queries.

Apache Cassandra[3], initially proposed by Lakshman and Malik [LM10], is a distributed NoSQL database written in Java and developed to combine Amazon *Dynamo* [DHJ$^+$07] distributed storage and replication techniques with Google *Bigtable* [CDG$^+$08] data and storage engine model. *Cassandra* employs a wide column data model. *Cassandra Query Language (CQL)* offers a similar model and syntax to SQL with statements to define and manipulate data. Data retrieval is more limited when compared to SQL, for example the ability to join data is absent. Cassandra does not provide ACID transactions with rollback or locking methods, opting instead for atomic, isolated, and durable transactions. Replication in *Cassandra* is done through a Multi-Master scheme with multiple coordinator nodes in charge of replicating the data items within a certain range and non-coordinator nodes acting as replicas. Cassandra offers an eventual/tunable consistency, allowing the client to select the level of consistency required for each session. Cassandra provides horizontal scalability by utilizing a hash algorithm to partition all data stored in the system. A storage engine capable of running in-memory only is not supported.

[3] https://cassandra.apache.org/

2.2.5 Document-oriented

As mentioned by Han et al. [HHLD11] and Jatana et al. [JPA^{+}12], similarly to key-value databases, the *document-oriented data model* also constitutes a "key-value" relationship. However, the distinction is made regarding the semantics of the value, referred to as a document, which instead of being encoded in a simple type, is serialized in some standard form such as XML, JSON, and BSON.

MongoDB[4] is a document oriented database implemented in C++, with BSON formatted documents as the basic unit of data. Each document consists of a set of key-value pairs that can vary from document to document — dynamic schemas. MongoDB has a distinct language model from SQL with a JSON-like syntax, which provides operations to manipulate documents (create, read, update and delete) and a similar level to relational databases of operators to query data from them. Nodes are able to execute server-side code implemented in JavaScript. MongoDB adheres to ACID principles for multi-document transactions. MongoDB supports a Primary-Backup replication scheme, with automated failover, where secondary nodes replicate the primary's data set. By default, the replication is done synchronously and MongoDB client sessions can ensure up to a causal level of consistency. In order to scale horizontally, MongoDB supports two methods of sharding: *Hashed Sharding* and *Ranged Sharding.* MongoDB nodes are also able to run on an in-memory only storage engine.

2.2.6 Summary

Relational databases are able to offer critical properties, such as ACID transactions, enforcement of a database schema, flexible query language and a strong level of consistency through its replication method. Transaction-oriented applications are able to leverage these properties effectively, while other use cases require a higher level of availability, horizontal scalability and performance. Non-relational databases are able to fulfill these needs by relaxing some properties, such as query language expressiveness, transaction semantics and consistency guarantees.

In sum, the choice of database type is determined by the expected workload access patterns, type of data to be stored and acceptable assurances. A comparison of the considered databases is summarized in Table 2.1.

[4] https://www.mongodb.com/

Table 2.1: Summary comparison of the studied databases.

Database Implementation	**PostgreSQL**	**Redis**	**Cassandra**	**MongoDB**
Chosen Database Model	Relational	Key-value	Wide column	Document
Implementation Language	C	C	Java	C++
Schema Required	yes	no	no	no
Defined Format Types	yes	partial	yes	yes
Server-Side Scripting	PSQL	Lua	no	JavaScript
Transactions	ACID	Atomic, Isolated and Durable Execution of Command Blocks	Row-Level Atomicity, Isolation and Durability	Multi-Document ACID
Concurrency	yes	yes	yes	yes
Durability	yes	yes	yes	yes
Replication Methods	Primary-Backup Replication	Primary-Backup Replication	Multi-Master Replication	Primary-Backup Replication
Consistency Guarantees	Strong Consistency	Eventual Consistency	Eventual/ Tunable Consistency	Eventual Consistency, Causal Consistency
CAP	CP	AP	AP	AP
Partitioning Methods	Table Partitioning (By range, list or hash)	Sharding (Hash Slot)	Sharding (Consistent Hashing)	Sharding (Hashed or Ranged)
In-Memory only support	no	yes	no	yes

2.3 Data Processing

CROSS relies on a significant volume of collected Wi-Fi and other signal observations. We will discuss two distinct data processing methods of producing useful information to subsequently integrate it in our operation: batch and stream.

2.3.1 Batch Processing

Batch processing is a method of processing large volumes of input data and producing some output data, at once. This is typically achieved through batch jobs, which can take a few minutes to several days. Batch jobs are often scheduled to run periodically, for example once a day. A concrete implementation of this technique is MapReduce [DG04], a programming model for parallel and distributed processing over large datasets.

2.3.2 Stream Processing

Batch processing assumes the input data is bounded, meaning that it is of a known and finite size. However, most systems require near real-time capabilities, since data might arrive at any moment and continue to grow progressively over time and they need to respond to it as it happens. A *stream* is a concept that encapsulates this notion of data that is made available gradually over time.

Stream processing is a method of continuously processing input data and producing some output data. A stream processor reacts to events as they occur, whereas a batch job operates on a fixed set of input data. More specifically, these events are generated by *producers* and can then be processed by *consumers*. To handle the storage and transmission of these streams, stream processing systems are usually built on top of a message broker. Producers write messages to the broker, and consumers receive them by reading them from the broker. Concrete implementations of stream processing frameworks are Apache Storm[5] and Apache Samza[6] [NPP$^+$17].

2.4 Architectures

Most modern data-sensitive systems demand *real-time data analytics*. The capability to react to events as they occur, for example in social media recommendation engines or in financial institution fraud detection algorithms, is simply not feasible with a dataset obtained through a data processing job executed the day before. These systems require asynchronous data trans-

[5] https://storm.apache.org/

[6] https://samza.apache.org/

formations with minimal delay. However, this should not sacrifice processing of historical data, meaning to reprocess past input data, which is crucial to cope with change in the system.

We now discuss two concrete scalable and fault-tolerant architectures - Lambda and Kappa - that are widely utilized to address the challenges outlined above, as well as their trade-offs.

2.4.1 Lambda Architecture

Marz proposed the "*Lambda Architecture*" [Mar11] in 2011 with the goal of achieving both real-time and historical data processing capabilities by combining both batch and stream methods. The Lambda architecture, as illustrated in Figure 2.2, is composed of three distinct layers: a *batch* layer, a *speed* layer and a *serving* layer. Captured data is continuously fed to the system as immutable sequences of records to both the batch and speed layers. The batch layer is responsible for storing the immutable, ever growing master dataset and recomputing a series of *batch views* which facilitate the computation of arbitrary queries over the dataset. Running batch jobs to maintain these precomputed views of the dataset takes a significant amount of time. Furthermore, any queries made during the precomputation process lack access to the data captured during this period. Therefore, to compensate for high latency updates the speed layer is responsible for incrementally computing a series of *real-time views* of recent data. Queries are handled by the serving layer against the merged results of both the batch and real-time views [WM15].

These capabilities increase the complexity with regards to code maintainability, since two separate complex distributed systems need to be indefinitely maintained. Both the batch and speed layers are required to be synchronized for the speed layer to correctly integrate the missing updates that occur during a batch job, resulting in increased computational time and effort [Lin17].

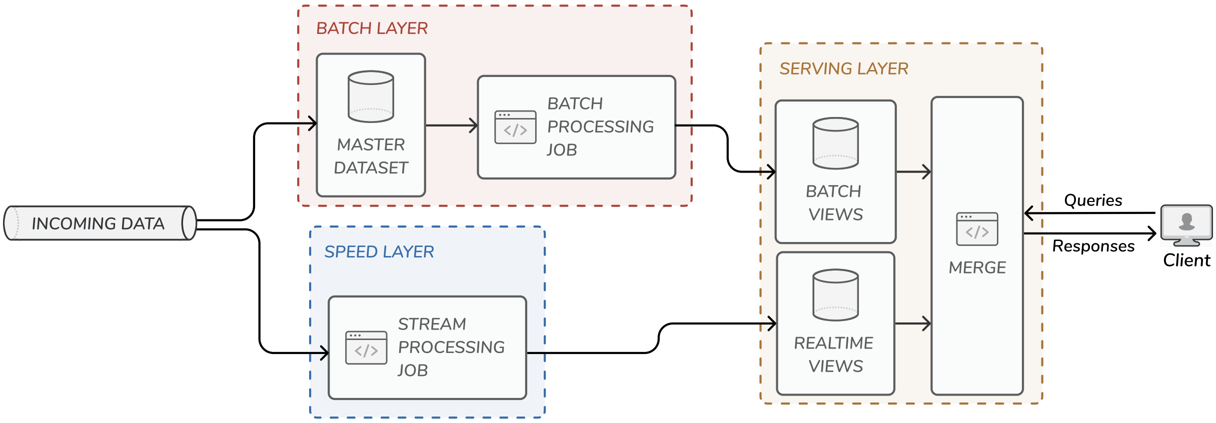

Figure 2.2: Overview of the Lambda data processing architecture.

2.4.2 Kappa Architecture

Kreps proposed the "*Kappa Architecture*" [Kre14] in 2014 to overcome the limitations of the Lambda Architecture. In the Kappa architecture there is no notion of batch. Every data is treated as a stream and therefore only a stream processing engine is required. As illustrated in Figure 2.3, it consists of two distinct layers: a *stream* processing layer and a *serving* layer. Data is captured and fed to the system as streams, similarly to the Lambda architecture, however due to the absence of the batch layer, it is simply directed to the stream processing layer. The stream processing layer is responsible for running the real-time data processing jobs. Queries are handled by the serving layer against the results from the stream processing layer. It is important to note, however, that data may still be reprocessed by simply *streaming* through historical data.

The Kappa architecture aims to achieve a "*general-purpose*" solution with both real-time and reprocessing capabilities without the added complexity of maintaining two separate systems. In this case, the trade-off is between latency/throughput and efficiency when reprocessing historical data. A low-latency stream processing engine is not going to outperform a high-throughput batch processing engine on a batch processing task built for that purpose [Lin17].

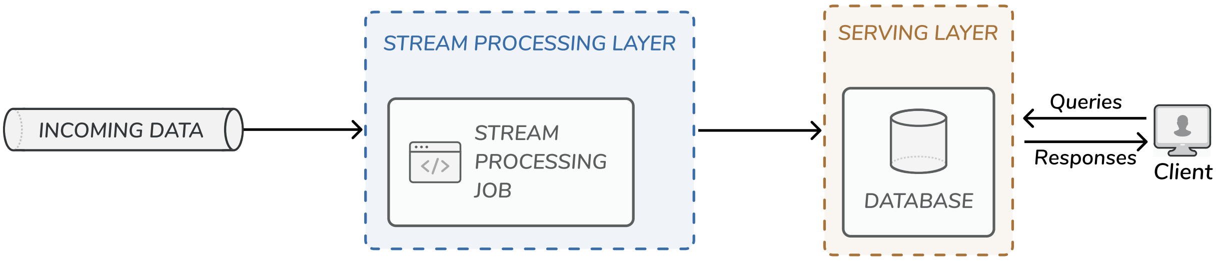

Figure 2.3: Overview of the Kappa data processing architecture.

2.5 Cloud Computing

Cloud computing has emerged as an appealing means of providing services over the Internet with the flexibility to provision additional resources as needed and employ a "*pay-as-you-go*" pricing model. Therefore, it is important to explore the virtualization technologies that support cloud computing, the cloud delivery and deployment models, as well as the offered container-based managed services.

As mentioned by Bernstein [Ber14] and Pahl [Pah15], virtualization technologies are at the foundation of cloud computing, whether via a hypervisor-based deployment with virtual machines (VMs) or a container-based deployment with containers. Due to their stronger isolation guarantees, VMs pioneered as the primary virtualization form on the cloud, however these guarantees impose various constraints, such as the requirement for full guest operating system (OS) images for each VM, which results in increased RAM and disk storage needs with poor startup performance.

2.5.1 Deployment Models

The NIST organization [MG11] defined the notion of cloud deployment model. Several deployment models are available for a provider's cloud infrastructure. The main distinction between each model is in the level of exclusivity granted to a cloud consumer, with respect to the cloud computing resources provided:

- **Private cloud**: A single organization has exclusive access to the infrastructure and the underlying computing resources. It can be managed by the consumer organization or by a third party, meaning it can be hosted on-site or by a hosting company;
- **Community cloud**: Shared infrastructure for a group of enterprises with similar mission objectives, security, privacy, and compliance policies. It, similarly to the previous model, can be administered by a group of companies or by a third party, being on-premise or off-premise;
- **Public cloud**: The general public has access to both the cloud infrastructure and the underlying resources. The public cloud is owned by an industrial group and run by a company that sells cloud services, catering to a wide range of clients;
- **Hybrid cloud**: The cloud infrastructure is made up of two or more clouds (private, community, or public) that are separate, yet linked by technology facilitating data and application portability.

2.5.2 Delivery Models

The Software-Platform-Infrastructure (SPI) model has been defined as a classification approach for cloud computing by NIST [MG11]. This scheme stands for the three main cloud-based services, offered by all the major providers:

- **Infrastructure as a Service (IaaS)**: The consumer has the ability to provision compute capacity, storage, network capacity and other key computing resources. The consumer is free to deploy and run any arbitrary software, which can include operating systems. The provider has the responsibility to manage and control the underlying cloud infrastructure;

- **Platform as a Service (PaaS)**: Customers are able to deploy already developed applications, with programming languages and tools supported by the provider, into the cloud infrastructure. The provider has the responsibility to manage and control the underlying cloud infrastructure;

- **Software as a Service (SaaS)**: The provider offers applications to users over the network.

Latest advancements in virtualization technology, such as containers, have also allowed the emergence of the *serverless* computing model, as described by Baldini et al. [BCC$^+$17] and Castro et al. [CIMS19]. Applications are broken down into small snippets of code that may be triggered and executed in an arbitrary cloud infrastructure. The term "*serverless*" comes from the shift in the focus of the developers away from the configuration, provisioning and management of servers to focus on the business logic. Naturally, the code is still executed on server machines.

The **Function as a Service (FaaS)** delivery model is the realization of *serverless*. Code logic is encapsulated in small stateless functions executed for a set period of time, after being triggered by an event such as an HTTP request. FaaS can be viewed as an evolution of PaaS, as it removes further customer control over the configuration of the execution environment, concealing complex components such as scaling policies. It also has a more fine-grained cost model, in which clients are only billed for actual code execution. However, the FaaS stateless model can lead to restrictive application programming and increase vendor lock-in.

A summary of the different responsibility concerns between the customer and the service provider across the distinct delivery models is shown in Table 2.2.

Table 2.2: Delivery model responsibility sharing between customer and service provider.

Delivery Model	**IaaS**	**PaaS**	**FaaS**	**SaaS**
Interface	Customer	Customer	Customer	Customer
Application Custom Code	Customer	Customer	Customer	Service Provider
Application Runtime	Customer	Service Provider	Service Provider	Service Provider
Operating System	Customer	Service Provider	Service Provider	Service Provider
Hypervisor	Service Provider	Service Provider	Service Provider	Service Provider
Computing Service	Service Provider	Service Provider	Service Provider	Service Provider
Storage Service	Service Provider	Service Provider	Service Provider	Service Provider
Network	Service Provider	Service Provider	Service Provider	Service Provider
Local Infrastructure	Service Provider	Service Provider	Service Provider	Service Provider

2.5.3 Container-based Managed Services

Application architectures composed of various services that share the underlying infrastructure, in the cloud, need a solution that addresses packaging, deployment, and portability concerns. Containers can satisfy these criteria. Docker expands on Linux Container (LXC) techniques, which use namespaces and control groups to isolate processes on a shared OS, by offering a systematic approach to packaging and deploying applications as portable containers while performing at a similar native level to make the most of the available computational resources.

However, as the usage of containers at a larger scale grows, so does the demand for a set of management tools, such as Docker Swarm and Kubernetes, able to oversee the networking between containers, the automation of their deployment, scaling policies, and the overall infrastructure operation. Cloud providers offer managed orchestration tools as a service, as a means to ease the use of these complex tools.

Ferreira and Sinnott [FS19] measured and evaluated CPU, memory, disk and network performance of manually deployed Kubernetes clusters and compared it against the performance of managed clusters deployed through Kubernetes-based cloud services. Specifically, they have considered the most popular service providers, and their Kubernetes-based cloud services: Amazon Elastic Container Service for Kubernetes (EKS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE). According to their experimental evaluations, the large percentage of the performance variations observed were tightly linked to the underlying resources delivered by the provider such as the VM type, storage type and network tier. Using managed cloud services, by themselves, does not result in a performance overhead.

2.6 Summary

In this Chapter, we showcased the initial CROSS prototype and some of its shortcomings in the generation and verification of time-bound location proofs. Additional data storage and processing components on the server-side are required to incorporate this feature in CROSS. Therefore, we analysed the main database types as well as a concrete implementation of each data model, discussed the different data processing modes and architectures which incorporate both components. Finally, we looked at cloud computing as a means of delivering services on demand.

Chapter 3

Implementation

In this Chapter, we describe CROSS City Cloud, the server-side implementation of the new version of the CROSS smart tourism application. The mobile application was led by a colleague working in the same project [Gra22]. The CROSS V1 architecture required several extensions to support the generation and verification of location proofs with time granularity. Additionally, we deployed the resulting cloud solution to a public cloud provider, alongside the integration of service operation related processes. With our work, CROSS services are available on demand for any client front-end application to consume, in particular the CROSS City mobile application.

In Section 3.1, we discuss the assumptions under which the new system was developed, as well as the requirements it must fulfill. In Section 3.2, we examine alternative architectures, and present an architectural overview of the extensions made to the data management layer of CROSS City, with a description of the functionality of each component and how components communicate and interact with one another. In Section 3.3, we select the proper cloud services for each architectural component, along with the reasoning behind each decision. We conclude this Chapter with Section 3.4, where we summarize the implementation details.

3.1 Assumptions and Requirements

CROSS City Cloud extends the functionality of the CROSS v1 prototype with increased dependability, therefore we retain some of its original assumptions while making some new ones:

- The system will be deployed in urban environments with a high density of diverse Wi-Fi networks, Bluetooth beacons and GSM cell-towers;

- The existence of GSM cell-towers with sufficient sky visibility for mobile devices to use GNSS;

- The system will be operated by *system operator* entities which are businesses directly in the tourism industry, or other organizations hired by them;
- Prior to the system going live, the system operators define the points of interest (such as emblematic city locations), touristic routes and corresponding rewards;
- Within the *epoch*, prior to the system going live, system operators collect sufficient network observations with multiple devices across the points of interest to identify the stable networks;
- At the end of a full *period*, we have sufficient network observation data collected by tourists to determine transient networks at each point of interest.

For the purpose of illustration, we will assume that the time windows for validating location proofs, as discussed in Section 2.1.3, are *epoch* of 1 week, *period* of 1 day and *span* of 1 minute.

We must still be able to guarantee that the initial CROSS v1 functional requirements are still met, including:

- **R1**: Rewards should only be given to people who are actually eligible to receive them;
- **R2**: System operators should be able to identify which users were rewarded with which rewards.

To adequately facilitate the generation and verification of time-bound location proofs, the system must also be able to fulfill the following functional requirements:

- **R3**: *Determine the stable network set by point of interest sliced by time*, which involves queries such as "What are the stable Wi-Fi SSIDs in this point of interest over the *epoch*";
- **R4**: *Determine the volatile network set by point of interest sliced by time*, which involves queries such as "What are the transient Wi-Fi SSIDs in this point of interest over the *span*".

In addition to the functional requirements, we established critical non-functional requirements that the system must be capable of satisfying:

- **Security**: Public clouds typically follow a shared responsibility model, as described in Section 2.5.2, thus security is a major concern. The infrastructure and communications between services needs to be protected against unauthorized use. Additionally, the confidentiality and integrity of the data must be preserved while stored persistently or in-transit;

- **Reliability**: The architected solution should tolerate crash faults and continue to function to achieve high availability;
- **Scalability**: The system should be capable to handle growing loads, by linearly adding more computational resources to meet that demand;
- **Elasticity**: The system should be able to provision or deprovision resources to meet the workload;
- **Monitoring**: To allow the detection of problems, the health status of each system component should be monitored;
- **Performance and Cost Efficiency**: The computing resources should be tuned for optimal performance while meeting the system requirements and operating at the lowest feasible price point;
- **Maintainability**: The system infrastructure and cloud services should be easy to manage and extend.

3.2 Architecture Extensions

In this section, we detail the extensions to the CROSS V1 prototype architecture, necessary to properly support time-bound location proofs, and discuss possible alternatives. Furthermore, we identify and mitigate conflicts with the original architectural components.

3.2.1 Data Management Layer

The CROSS prototype user flow always necessitates client communication with the back-end before and after a trip. Before starting a specific trip, the client application fetches the catalog of locations and possible itineraries. During the trip, the client application logs the visits to each point of interest while sensing Wi-Fi signals, and either stores these locally (offline connection) or publishes them to the API server (online connection) as it senses. At the end of the trip, the application submits all of the collected information, that has not been submitted during the trip, to the back-end and claims each point of interest visited. It is important to note that the start and end of a trip might coincide with the beginning and end of a day/*period*, but it is not guaranteed.

Based on the expected user flow, one crucial question emerges: Is there a necessity for real-time data analytics? If the answer to this question is negative, then a minimal Lambda architecture with solely batch processing is sufficient to provide us offline data processing capabilities. However, if the answer to this question is positive, then either the Lambda or Kappa architectures are potential solutions. Additionally, the Kappa architecture is preferable over the Lambda architecture due to the lower level of complexity to maintain it, as mentioned in Section 2.4.2, and our ability to address the correctness of the solution by ensuring that the stream processing system delivers a semantic guarantee as strong as the one that is provided by the batch system.

Minimal Lambda Architecture Rationale

If we consider that the start and end of a trip typically coincide with the beginning and end of a day/*period*, then we can argue that real-time capabilities are not required in this particular scenario. The process of generating a time-bound location proof is highly dependent on the integration of the complete set of network observations collected by the greatest number of co-located witnesses within a *period*, as described in Section 2.1.3. Ideally, with real-time capabilities we should be able to ingest, process and integrate the observations into the operating dataset as soon as they happen, however this does not ensure that we have received all observations that have occurred up until that point in time. By purposefully delaying location proof requests and integrating all of the observations received in a period of 24 hours, through offline data processing, we do not solve this issue, but we would have an equal chance of determining the most encompassing volatile network set. Figure 3.1 is a data flow diagram showcasing the process of ingesting submitted network observations collected by tourists in an *Observations Storage* data store, and the batch transformation and integration of network observations in an *Serving Storage* data store to serve stable and volatile network queries.

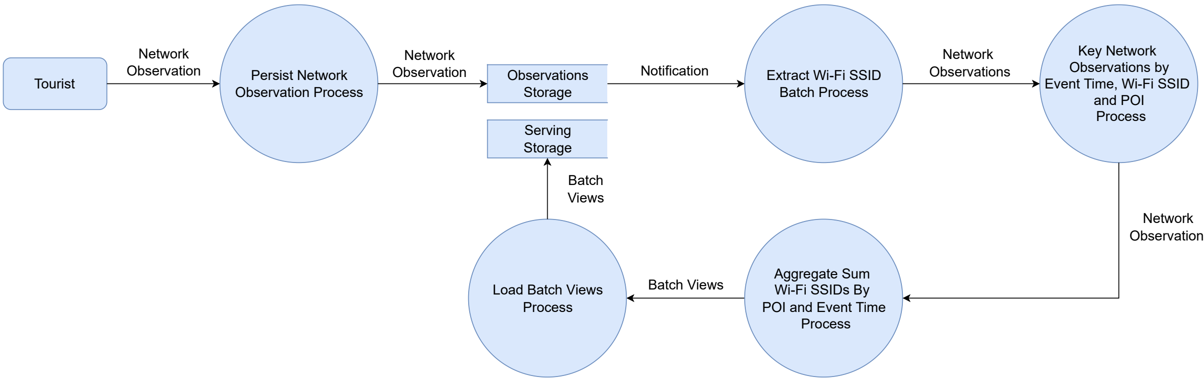
Tourist
Network Observation
Persist Network Observation Process
Network Observation
Observations Storage
Serving Storage
Notification
Extract Wi-Fi SSID Batch Process
Network Observations
Key Network Observations by Event Time, Wi-Fi SSID and POI Process
Network Observations
Batch Views
Load Batch Views Process
Batch Views
Aggregate Sum Wi-Fi SSIDs By POI and Event Time Process

Figure 3.1: Data flow diagram detailing the network observations data ingestion, processing and integration for the minimal Lambda architecture.

Kappa Architecture Rationale

The minimal Lambda architecture solution detailed in the previous Section 3.2.1 relies on the assumption that the start and end of a trip typically coincide with the beginning and end of a day/*period*. Alongside this assumption, it must also be assumed that during the day/*period* no location claim request is made for that particular *period*, or at the very least that this user behavior is expected to be the most common one. However, if we consider that real-time capabilities are necessary, then we are able to relax these assumptions. Network observations published by users can be persisted as soon as they are received, as streams, then processed and integrated into the operating dataset in real-time. These capabilities ensure that we produce responses in a timely manner through low latency updates, even with incomplete data. Solely with offline or batch processing, any location proof requests made during a *period* would necessarily have to be purposely delayed until a *period* reached its completion or multiple high latency batch jobs would have to be triggered during the *period*, for us to be able to fulfill those requests. Furthermore, real-time capabilities would allow us to both extend the solution with logic to potentially react and predict user behavior in real-time and support latency sensitive clients. Figure 3.2 is a data flow diagram showcasing the process of ingesting published network observations collected by tourists, as streams, in an *Observations Stream Storage* data store, and the stream processing job processes to integrate network observations in a *Serving Storage* data store to serve stable and volatile network queries.

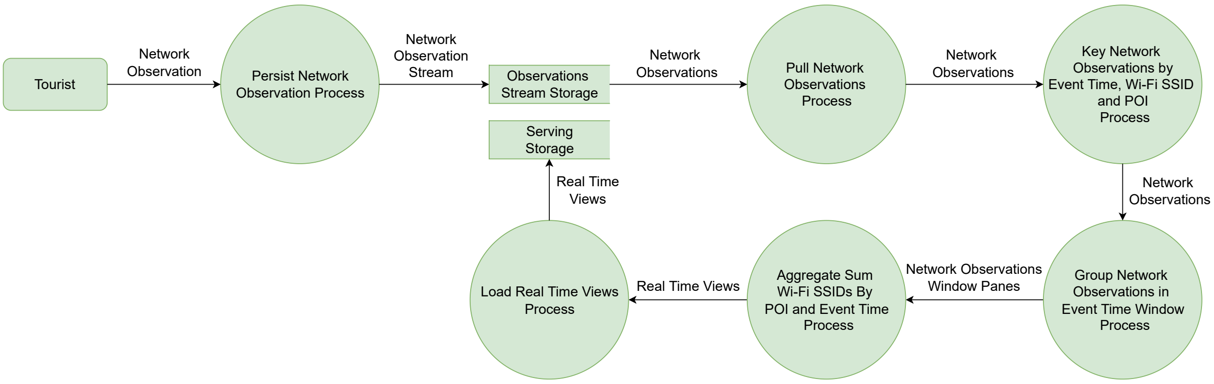
Tourist
Network Observation
Persist Network Observation Process
Network Observation Stream
Observations Stream Storage
Network Observations
Pull Network Observations Process
Network Observations
Key Network Observations by Event Time, Wi-Fi SSID and POI Process
Network Observations
Group Network Observations in Event Time Window Process
Network Observations Window Panes
Aggregate Sum Wi-Fi SSIDs By POI and Event Time Process
Real Time Views
Load Real Time Views Process
Real Time Views
Serving Storage

Figure 3.2: Data flow diagram detailing the network observations data ingestion, processing and integration for the Kappa Architecture.

Final Architecture

With the additional extension, CROSS City Cloud needs to offer the following services:

- Serve domain touristic and user information requests (insert, read, update and delete);

- Authenticate users;

- Assign rewards;

- Verify location proofs;

- Store and validate network observations;

- Process network observations;

- Fetch stable and volatile network set data.

Based on the detailed rationales and operations, we extend the CROSS architecture data management layer with three distinct layers: a *domain* layer, a *stream* layer and a *serving* layer. The domain layer will be responsible for storing all of the entity relations and their corresponding related data, such as the user information, points of interest and tourism routes. Any queries related to the domain data will be handled directly by the domain layer. The stream layer stores the raw streams of Wi-Fi signal observation data as atomic facts, these facts are kept immutable and true within a given *epoch* time window through the use of publish time timestamps. With this fact-based model, since no data is altered and always true as per the time of its addition, we are able to recompute views from historical data, which is particularly useful in the event of any possible requirements change. Additionally, the stream layer is responsible for executing stream processing jobs that will produce stream views containing precomputed aggregated results to assist stable or volatile set queries. The serving layer indexes the results processed the stream layer, to serve the stable or volatile set query requests. Since processing is done asynchronously on the stream layer, the serving layer is only required to fetch the sets while handling client query requests, avoiding extra computation work.

Figure 3.3 illustrates the CROSS v1 prototype server-side architecture to contrast with Figure 3.4 which illustrates the resulting CROSS server-side architecture with the extension described previously.

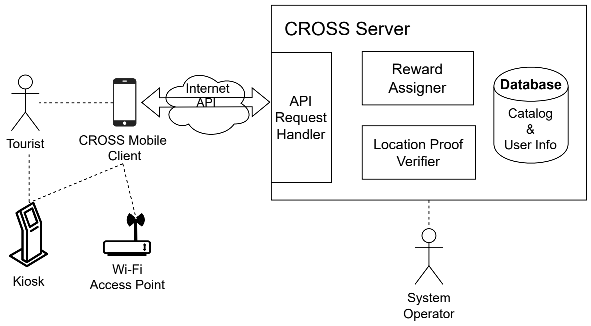

Figure 3.3: Overview of the CROSS v1 prototype server-side architecture.

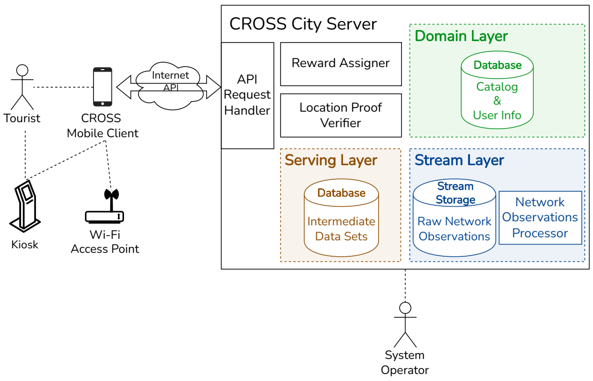

Figure 3.4: Overview of the extensions to the CROSS server-side architecture.

3.2.2 Network Observation Data Collection

The diversity of signals at each point of interest is leveraged as an ad-hoc witness to prove the location and time of visit, as described in Section 2.1.3. CROSS City Cloud specifically utilizes the stable networks to prove the location and the volatile networks to prove the time of visit. Thus, networks observed by tourists must be continuously integrated into the operational dataset. Nonetheless, we need to ensure that only valid network observations are integrated.

With this in mind, the CROSS City Cloud lifecycle is split into two stages: **Pre-Live** and **Live**. The Pre-Live stage is a finite time interval with a total duration equal to the system *epoch*, referred to as $epoch_0$. Throughout the Pre-Live stage, only trustworthy entities such as the system operators submit network observations of each existing point of interest, with the goal of deriving the initial stable network sets. Since the system trusts the system operators, the verification of their submissions is not required. The Live stage is a sequence of *epoch* intervals, with the initial one named $epoch_1$. Throughout the Live stage, untrusted entities interact through trip submissions, as depicted in the UML sequence diagram of the protocol (Figure 3.5). Tourists/Provers are meant to complete trips across multiple point of interest visits and collect the networks observed. Each visit (location claim) contains a set of evidences (network observations) which are validated against the claimed point of interest stable network set of the former epoch. For $epoch_n$, the stable network set of $epoch_{n-1}$ is used for validation. Only if the claimed point of interest confidence threshold is fulfilled does the visit get accepted and its network observations are incorporated in the system.

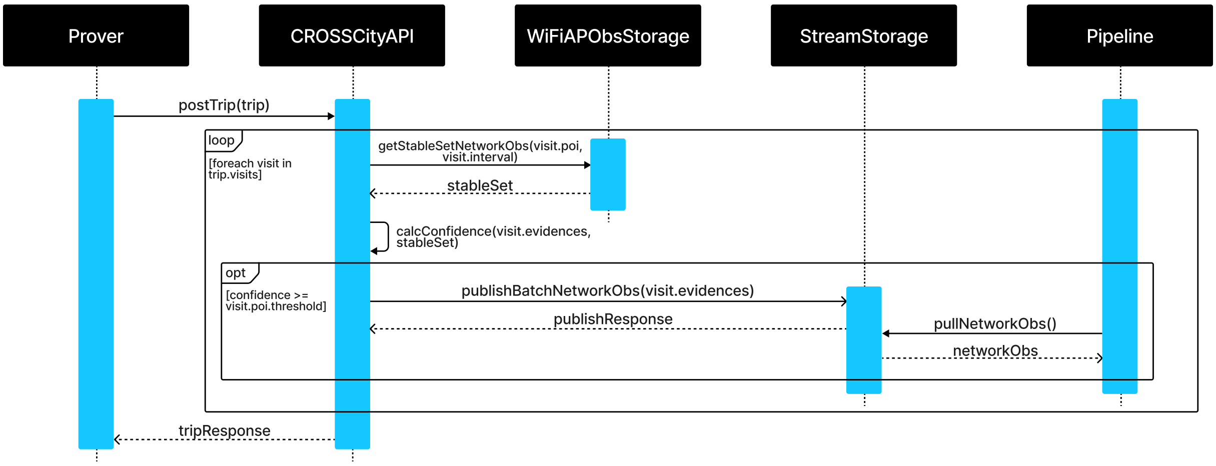
Prover
CROSSCityAPI
WiFiAPObsStorage
StreamStorage
Pipeline
postTrip(trip)
loop
[foreach visit in trip.visits]
getStableSetNetworkObs(visit.poi, visit.interval)
stableSet
calcConfidence(visit.evidences, stableSet)
opt
[confidence >= visit.poi.threshold]
publishBatchNetworkObs(visit.evidences)
publishResponse
pullNetworkObs()
networkObs
tripResponse

Figure 3.5: UML sequence diagram of the Prover/Tourist entity network observation publish protocol.

In sum, the Pre-Live stage represents the $epoch_0$ and has the main purpose of producing the genesis stable network sets of each point of interest, via data collection scavenged by system operator entities. The Live stage reflects the subsequent *epochs* and both stable and volatile network sets of each point of interest are produced, through the data collection provided by prover tourists. Figure 3.6 is a UML timing diagram illustrating the state change and the aforementioned entity interaction.

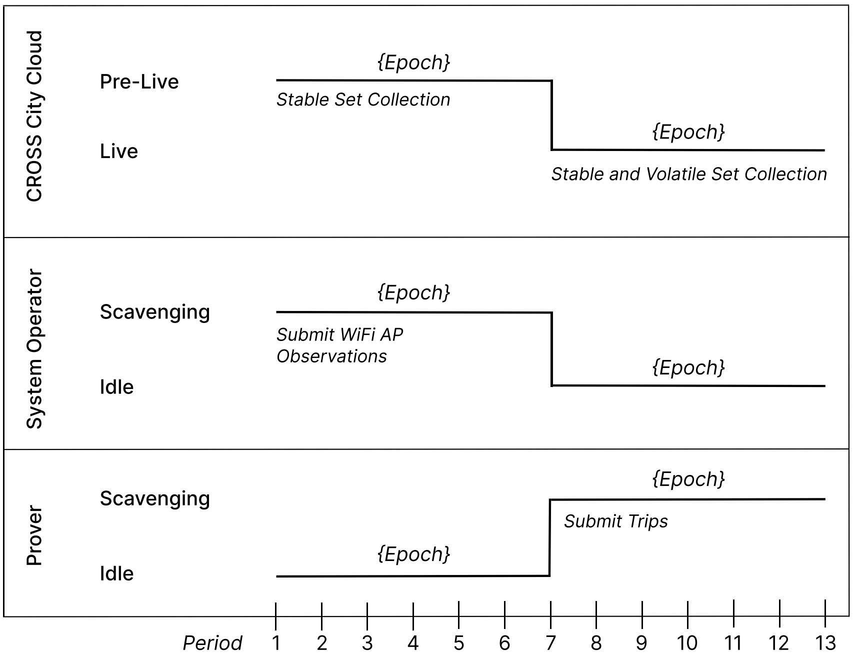

Figure 3.6: UML timing diagram of the CROSS City Cloud lifecycle across $epoch_0$ and $epoch_1$ with the system operator and prover entity lifelines.

3.2.3 Intermediate Network Observation Set Computation

The sole goal of the stream layer is to produce network observation views to be queried efficiently, in a low-latency manner. Hence, we are adopting an *incremental computation* approach over *recomputation*, avoiding the execution of our function logic over the entire set of observations. To be efficient, the views should contain intermediate results of the expected queries: *Most observed networks, over an epoch, for a given point of interest* (stable network set) and *Least observed networks in a span interval, over a period, for a given point of interest* (volatile network set).

The key idea of the intermediate views is to maintain a count of the number of observations per network at each point of interest, for the most encompassing range of time. Note that the usage of larger time intervals increases query performance by trading-off proof validation accuracy. We now reason about the ideal time interval for each set and the kind of window used for grouping (tumbling for fixed size non-overlapping time intervals and hopping for fixed size scheduled overlapping intervals [Kle17]). Stable network sets are queried within an *epoch*; since an *epoch* must always encompass a *period*, the minimum time window granularity is a *period*. Volatile network sets are queried within a *span*, and each of the possible span time windows encompasses smaller intervals of span's greatest common divisor size. For example let the $spans = \{15\,min, 10\,min, 5\,min\}$, given any *span* interval with size equal to one of the *spans*, it can be represented as a union of 5 min (the greatest common divisor) intervals. We leverage the fact that the set of spans are known before going live to compute every possible time interval of a span's greatest common divisor size with minute granularity, during a *period*, by aggregating network observations into one minute periodic hopping time windows. This computation is feasible as it results in, at most, 1440 windows (there are $24 * 60 = 1440$ *minutes* in a day) during computation time. Figure 3.7 represents the pipeline, each network observation is first pulled from the stream layer storage component, then aggregated in two separate tumbling and hopping windows, based on its publish time (event time), with size equal to the *period* and the greatest common divisor of the *spans*, then keyed and summed per point of interest and BSSID, and finally written to the serving layer. We also studied the possibility of computing each interval using hopping windows of each span's size, as this would result in a minor computation overhead of additional 1440 windows per span interval, however due to the relatively low size of a *period*, this alternative did not yield any gains over our solution.

3.2.4 Stable and Volatile Network Observation Set Computation

Both stable and volatile network set views are persisted in the serving layer and partitioned by point of interest and *period*, since the queries are expected to be tied to a particular point

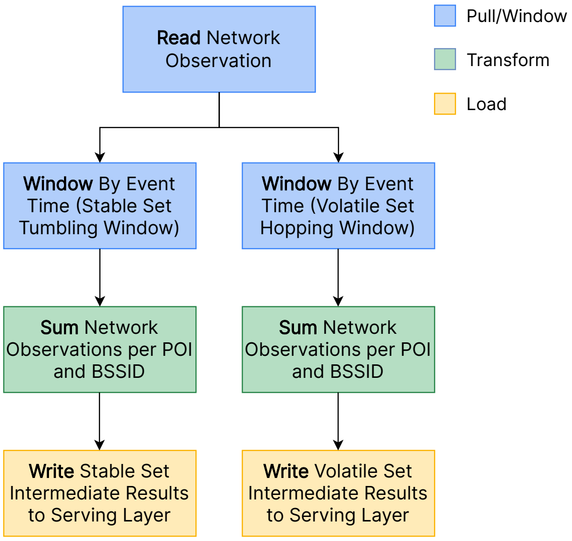
Pull/Window
Transform
Load
Read Network Observation
Window By Event Time (Stable Set Tumbling Window)
Window By Event Time (Volatile Set Hopping Window)
Sum Network Observations per POI and BSSID
Sum Network Observations per POI and BSSID
Write Stable Set Intermediate Results to Serving Layer
Write Volatile Set Intermediate Results to Serving Layer

Figure 3.7: CROSS City Cloud pipeline for producing intermediate stable and volatile network sets.

of interest and require, at most, a *period* worth of data. Hence, the usage of the horizontal partitioning method is an efficient way to store these specific views. Each record in the view maintains the number of observations for a particular network within a time interval, as detailed in Table 3.1. This method of caching is similar to the concept of materialized view which commonly is a copy that contains the results of a query [Kle17].

Table 3.1: Intermediate network observation set view fields.

Field	Description
poi_id	Point Of Interest
bssid	Collected Network
start_time	Initial Timestamp of the Collection Time Interval
end_time	Final Timestamp of the Collection Time Interval
count	Number of Observations for the Collected Network

As soon as an *epoch* is completed the *period* intermediate stable set views, that comprise the *epoch*, are used to produce a materialized view containing the top 10% observed networks over that *epoch* (Listing 3.1), off the critical path. Network observations from past *epochs* are expected to remain unchanged, thus the creation of an additional materialized view significantly improves the efficiency when accessing a stable set. Furthermore, volatile set queries utilize the intermediate volatile set views and the stable set materialized view to filter the top 10% observed networks of the previous *epoch* and retrieve the bottom 10% observed networks within the claimed time interval (Listing 3.2). We are considering 10% as the threshold value, nonetheless this remains a configurable parameter.

Listing 3.1: 1 week *Epoch* Stable Set Query - 2022-07-29 to 2022-08-04

```
-- Sum network observations for the epoch
WITH bssid_total_count AS (
    SELECT bssid, SUM(count) as total_count
    FROM (SELECT bssid, count FROM ‘intermediate_stable_set_2022-07-29‘ UNION ALL ... SELECT
        bssid, count FROM ‘intermediate_stable_set_2022-08-04‘)
    GROUP BY bssid)

-- Stable set for the epoch (top 10% observed)
SELECT bssid, total_count
FROM bssid_total_count
WHERE total_count >= (SELECT PERCENTILE(total_count, 0.9) FROM (SELECT * FROM
    bssid_total_count) LIMIT 1)
```

Listing 3.2: 10 Minute *Span* Volatile Set Query - 14:30:00 to 14:40:00

```
-- Sum network observations for the claimed span interval, and filter the stable set
WITH bssid_total_count AS (
  SELECT bssid, SUM(count) as total_count
  FROM ‘intermediate_volatile_set_table‘
  WHERE bssid NOT IN (SELECT bssid from ‘stable_set_table‘) AND (start_time = 14:30:00 OR
      start_time = 14:35:00)
  GROUP BY bssid)

-- Volatile set for the claimed span interval (bottom 10% observed)
SELECT bssid
FROM bssid_total_count
WHERE total_count <= (SELECT PERCENTILE(total_count, 0.1) FROM (SELECT * FROM
    bssid_total_count) LIMIT 1)
```

3.2.5 Catalog Immutability

Whenever extending or modifying any software architecture we should be aware of possible conflicts with existing components, to swiftly identify and mitigate them.

The stream layer, in our extension, sums network observations by their point of interest and event time, which in turn produces the intermediate results stored in the serving layer. As one might expect, the points of interest in the stable and volatile set views must exist in the domain layer of CROSS, therefore these intermediate results depend on the domain layer. More specifically, there is a dependence on the database module storing the domain data related to the points of interest and tourism routes, referred to as the *catalog*. In essence, this problem arises only whenever there is data inconsistency between the data in the catalog and the serving layer, such as the existence of different points of interest or routes, which can be solved by enforcing immutability in the catalog. By timestamping each tourist collected network observation and keeping each as the truth only within the *epoch* it originated, we are able to enforce immutability in the stream layer. Similarly, once the points of interest and tourism routes are created we are able to version and timestamp them, preventing changes to them and enforcing immutability of the catalog, within an *epoch*. This method guarantees that time-bound location claims are ensured to be true within the *epoch* claimed based on both the domain and serving layer.

3.3 Cloud Deployment

In this section, we assess the proper cloud offerings for each architectural component of the CROSS City Cloud architecture, detailed in Section 3.2.1. Furthermore, we discuss the numerous additional changes that were required, to fulfill the criteria enumerated in Section 3.1.

3.3.1 Base Deployment

Each one of the three layers of the CROSS City Cloud architecture has a different set of components which must fulfill a different set of requirements. The decision of each component's cloud service is dependent on the service that satisfies the greatest amount of requirements, without the need to be modified.

Domain Layer

When abstracting the implementation details, the domain layer is comprised of two primitive components: a compute (CROSS API Server) and a database (CROSS Domain Database).

Starting off with the CROSS API Server component, in the CROSS prototype, the client mobile application communicated with the back-end services through this component. It was a REST API (Application Programming Interface) exchanging JSON payloads. The code was implemented in Go and communication was done over HTTPS. HTTP is an application-layer protocol supported on reliable TCP connections designed for web communications [Mai19].

Due to maintainability reasons, specifically to address the evolvability and documentation of the code base, the CROSS API Server component was redeveloped in the Java programming language with Maven[1] as the software project management tool (build and dependency management automation). This API Server component remains the entrypoint for the communication between the client and the back-end services, containing all of the necessary handlers to handle domain data, authentication, reward assignment, and location proof validation requests. The

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

interface still follows the REST (REpresentational State Transfer) software architectural style for web services, which defines a set of constraints restricting the protocol used in Client-Server architectures with regards to the possible requests and responses. REST principles lead to a stateless application layer protocol, which helps us assure some of the requirements for our solution, such as performance, scalability, maintainability, and reliability. In stateless protocols, there is no concept of sessions and each request contains all the information necessary to be processed. This is useful in contexts where the services can be decoupled and scaled out. As for the implementation of this component, the Jersey 3.0 reference implementation[2] was used to support the Jakarta RESTful Web Services 4.0 specification[3] (JAX-RS) which eases and standardizes the development of web services according to the REST architectural pattern with Java annotations added to the source code. Additionally, Jersey helps to expose data in a variety of representation media types and abstracts away the low-level details of the client-server communication. GlassFish[4] is used as the application server with Grizzly[5] web server components. GlassFish is an open-source Jakarta EE platform application server project. Project Grizzly is an HTTP server framework, pure Java web service built using the non-blocking I/O (NIO) API for improved performance and scalability. The redevelopment was also used as an opportunity to ensure that the API server is compliant and uniform with other existing SureThing projects with regards to the use of the format protocol buffers for encoding data. Protocol buffers[6], originally proposed by Google, is an open-source language-neutral, platform-neutral, extensible mechanism for serializing structured data, allowing us to structurally serialize (binary encoding) our typed data across languages.

The API was documented following the OpenAPI[7] specification, which is a language-agnostic interface for REST APIs that allows current and future *system operators* to discover and understand the capabilities of the CROSS City Cloud services. The specification defines the expected requests and responses without requiring full access to the source code, through human-readable documentation about both the data and API specification. In pursuance of automatic generation of the OpenAPI description from the CROSS API Protocol Buffer service definitions, we have decided to leverage a third-party plugin, developed by Google, for the Protocol Buffer Compiler named “gnostic protoc-gen-openapi”[8]. This plugin allows the conversion of OpenAPI descriptions to and from equivalent Protocol Buffer representations. The CROSS API Protocol

[2] https://eclipse-ee4j.github.io/jersey/
[3] https://jakarta.ee/specifications/restful-ws/4.0/
[4] https://javaee.github.io/glassfish/
[5] https://javaee.github.io/grizzly/httpserverframework.html
[6] https://developers.google.com/protocol-buffers
[7] https://swagger.io/specification/
[8] https://github.com/google/gnostic/tree/main/cmd/protoc-gen-openapi

Buffer service definitions follow the Google APIs Guidelines[9], which involves providing HTTP definitions for each RPC defined, for compiling purposes.

Based on the goal of pushing as many infrastructure concerns as possible to the cloud provider, the most adequate delivery models in this scenario would be FaaS or PaaS. Any existent cloud compute offering, such as Google Cloud Platform (GCP) Cloud Functions or App Engine, Amazon Web Services (AWS) Lambda or Elastic Beanstalk and Azure Functions or App Service, is able to serve this kind of RESTful service. More specifically, Google Cloud offers the Google App Engine Platform, which is a fully managed cloud computing PaaS with built-in services, such as auto-scale based on demand and load-balancing, and fits most of our set criteria. Nonetheless, we want to mitigate cloud provider lock-in as much as possible, have a greater control over our scalability policies and node configuration flexibility, as well as uniformize future service deployments by using the same interface regardless of cloud choice. As discussed in Section 2.5.3, Kubernetes-based cloud services offer managed orchestration tools as a service providing a complete control over every aspect of container orchestration, from networking, to storage, and observability over each component. With this in mind, the CROSS REST API Server can be packaged as a Docker container and deployed to a Kubernetes cluster, such as Google Kubernetes Engine (GKE), as a service to be exposed to connections outside of the cluster. Based on this cloud deployment decision for the CROSS API server, some of our initial requirements are promptly fulfilled, where as others require further modifications to be assured:

- *Scalability* - If we describe load as the rate of requests made to the server, then by having a stateless application layer protocol derived from the enforcement of REST principles, we address the scalability aspect of this component. Moreover, services can be decoupled and scaled out independently, which can be leveraged with the use of a load balancer. Both scaling out and up can be attained within our Kubernetes deployment configuration. Scaling out is achieved by adding additional replicas and scaling up can be defined by requesting more from a resource such as CPU and memory within the set GKE node limits used. A load balancer is not obtained by simply using GKE, therefore this will be addressed in the following Section as a “deployment enhancement” to this component;

- *Elasticity* - The base deployment of this component is not inherently elastic with the use of GKE and therefore requires further modifications to fulfil this requirement;

- *Reliability* - With the use of Kubernetes, we are able to set the level of replication (number

[9] https://google.aip.dev/127

of identical pods) for this specific component, which when paired with the fact that the application layer protocol is stateless, high availability is assured. Fault tolerance at the pod level can be achieved with the use of a liveness probe, which periodically sends an HTTP request to a specific endpoint and waits for the response within a certain delay, if the threshold delay is surpassed without a response the pod is restarted, to overcome the failure, i.e. transition to a failure free state;

- *Maintainability* - This requirement is intrinsically satisfied with our decision to use Maven for build and dependency management automation, and with the enforcement of REST guidelines leading to decoupled services. Additionally, the API documentation generation, with the use of the OpenAPI specification, also contributes to the satisfaction of this requirement;

- *Performance* - Each GKE cluster is comprised of a node pool with variable node resource specifications set by us. Therefore, from a resource performance standpoint, by using GKE we have available, by default, a plethora of Google Cloud instance types. Furthermore, the derived stateless application layer protocol, from the enforcement of REST principles, contributes to assuring this requirement;

- *Security* - With regards to security, two major concerns arise: secure network communications between client and server through the TLS protocol, and secure access to the CROSS API server workload resources, in the GKE cluster, with authentication and authorization. Both concerns are not satisfied by default and thus require modifications detailed in the subsequent Section.

In the CROSS City Cloud deployment, the database component of the domain layer remains responsible for storing both user and tourism related data, similarly to the CROSS prototype. User information is used to serve the user authentication service, as well as provide specific information of each user account, regarding their trip history and rewards received. Tourism information is comprised of the available tourism routes, points of interest and possible rewards, referred to as the *catalog*. Due to the level of relation between the data described and the degree of query expressiveness required, we maintained the decision to use a relational data model and PostgreSQL as the specific relational database, since other alternatives were formerly discussed in the prototype [Mai19].

The deployment of the database could be attained through fully managed services such as Google Cloud SQL, Amazon RDS and Azure Database for PostgreSQL. However, considering the use of GKE for the deployment of the CROSS API server, we can leverage cluster multitenancy as a means to be more cost effective, while maintaining a similar level of management. Let us review the list of requirements concerning the database component with the deployment decision in consideration:

- *Scalability* - Similarly to what was mentioned with regards to the CROSS API Server, both scaling up can be attained within the Kubernetes deployment configuration. Scaling up can be defined by requesting more from a resource such as CPU and memory within the set GKE node limits used, which are also configured based on the instance types provided by Google Cloud or a custom one. In contrast to the CROSS API Server component, scaling out can not by achieved by simply adding additional replicas, due to the *Primary-Standby* replication method used by PostgreSQL, as detailed in Appendix A.1. Either synchronous or asynchronous replication between the primary and the read replicas (separate deployments) would need to be setup. While in our configuration replication is performed at the disk level;

- *Elasticity* - Identically to the CROSS API Server deployment, the database component base deployment is not elastic, by default. Moreover, as mentioned previously due to the *Primary-Standby* replication method utilized by PostgreSQL, replication between the primary deployment and the read replicas deployment would need to be established first. After that, elasticity within the read replicas would be satisfied;

- *Reliability* - The combination of a GKE regional cluster (multi-zone replicated with multiple masters - one per zone in the region) and a regional persistent disk provides durable storage and synchronous replication of data between two zones, in the same region, guaranteeing that an outage in a single zone does not make the database unavailable. Although PostgreSQL does not provide automatic failover, as described in Appendix A.1, this can be mitigated with the use of a Kubernetes Deployment workload resource. In a Kubernetes Deployment workload resource we can describe the desired state (one database Kubernetes pod), and the Deployment controller ensures that the actual state matches the described desired state;

- *Performance* - Each GKE cluster is comprised of a node pool with variable node resource specifications. Therefore, from a resource performance, such as CPU and memory, standpoint, we have available, by default, a plethora of Google Cloud instance types. As for the persistent disk performance achieved, the persistent volume claim used by the database instance is based on the regional persistent disk types available, which can be resized to obtain more throughput and IOPS.

Stream Layer

The stream layer is comprised of two primitive components: data ingestion and data processing.

The *data ingestion* component has the purpose of ingesting network observation events, published by clients through the CROSS API, for streaming into the data processing component. In the current architecture, from a publish/subscribe model standpoint, the CROSS API Server pods are considered the producers and the processing pipeline worker the consumers. Thus, support for both multiple producers and multiple consumers should be satisfied.

We could either implement a messaging system with direct communication between producers and consumers (*brokerless*) or utilize intermediary nodes (*brokers*). The application code in brokerless systems typically has to be aware of the risk of message loss, even though they function effectively in the scenarios for which they are intended. Its tolerable faults are relatively constrained, despite the fact that the protocols identify and resend packets that have been lost in the network. It is typically assumed that producers and consumers are always online, as a consequence a consumer may miss communications received while it was unavailable. Some protocols let the producer reattempt a failed message delivery, however this strategy may fail if the producer crashes and loses the buffer of messages it was intended to try again with. A message broker is essentially a type of database that is designed specifically to handle message streams. Consumers read the messages that producers publish to a broker topic, after they have been written to the broker. By centralizing the data in the broker, these systems can more readily withstand clients who connect, disconnect or crash. Moreover, the durability assurance is thereby transferred to the broker.

Consumers are typically asynchronous as a result of queueing, for example, when a producer delivers a message, it typically simply waits for the broker to validate that it has buffered the message and does not wait for consumers to digest the message. The delivery to customers will take place at some unspecified period in the future. For our specific use case, a message broker fits more our criteria. A plethora of message brokers are available, such as RabbitMQ[10], ActiveMQ[11], Apache Kafka[12], Azure Service Bus[13] and Google Cloud Pub/Sub[14]. The decision between the message brokers lies on our application requirements. Specifically, we have previously mentioned that the message broker must support multiple producers and consumers on the same topic. Message ordering is not necessary, since our intermediate results are aggregated on event-time and are not dependent on the delivery order. Replaying previous events is re-

[10] https://www.rabbitmq.com/

[11] https://activemq.apache.org/

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

[13] https://azure.microsoft.com/en-us/services/service-bus/

[14] https://cloud.google.com/pubsub

quired, thus message retention with period of an *epoch* is necessary. To avoid the loss of any network observations and duplicates from retries, *at-least-once* delivery paired with *exactly-once processing semantics* must be guaranteed.

A *pull-based* model, where the consumer pulls messages from the broker is preferable over a *push-based* model. Push-based systems struggle to handle a diversity of consumers, since the broker sets the data transmission rate. In a push system, the consumer can become overwhelmed when its rate of consumption falls below the pace of production, which is bad because the aim is often for the customer to be able to consume at the greatest rate feasible for that specific consumer. In a pull-based system the consumer just lags behind and catches up when it can.

A contract should be able to be enforced at the broker level, between the producer and consumer, specifying both the format of the network observation messages, as well as their encoding. Moreover, this contract should support the Protocol Buffer format, considering this is the format of choice at the domain layer.

Based on the fulfilment of the detailed requirements, Google Cloud Pub/Sub was chosen as the message broker/data ingestion component. Let us assess the set of requirements for this component:

- *Scalability* - Pub/Sub attains horizontal scalability through per-message parallelism, rather than partition-based messaging. Individual messages are *leased* by Pub/Sub to topic subscribers, and it then keeps track of whether each message is properly digested;

- *Elasticity* - Pub/Sub guarantees automatic capacity management with both auto-scaling and auto-provisioning;

- *Reliability* - Pub/Sub ensures that all data is replicated synchronously to at least two zones and best-effort replicated in a third zone;

- *At-least-once delivery* - Publishers push messages to the brokers, and synchronously wait for the broker to confirm that it has buffered the message. Then, messages get cross-zone replicated, and tracked individually on a per-message acknowledgement to ensure at-least-once delivery;

- *Message Retention* - A Pub/Sub topic defaults to discarding messages once they have been acknowledged by every subscriber attached to the topic. Nevertheless, a topic can be setup with message retention, which enables any subscription attached to the topic to seek back in time and replay previously acknowledged messages. A topic can retain published messages for a maximum of 31 days, which is within our *epoch* time window requirement of 7 days;

- *Protocol Buffer Schema* - Pub/Sub supports the creation of Protocol Buffer schemas. A schema establishes a contract between a publisher and a subscriber that Pub/Sub will enforce by defining the format that the message data fields must adhere to. Schemas are versioned resources, and assigned to specific Pub/Sub topics.

Regarding the *data processing* component, responsible for processing the network observations ingested and producing the intermediate results for the stable and volatile set queries, two decisions must be made. First, concerning the processing engine used, and second, with regards to its cloud deployment. The pipeline is meant to aggregate network observations on event-time in two separate tumbling and hopping windows, corresponding to stable and volatile set windows. Moreover, we should expect late and duplicate network observations. With this in mind, the processing engine must be able to compute aggregations on event-time, not processing-time, automatically manage state and resources, elastically scale the system and also be fault-tolerant. Additionally, since the data ingestion component solely assures at-least-once delivery, this component must guarantee *exactly-once processing semantics*. Furthermore, as an optional requirement the engine should likewise be capable of batch processing, if required. Note that some of these requirements are only assured when the engine is deployed to the respective cloud service. Common options that satisfy each requirement and provide Java Software Development Kits (SDKs) to maintain the code base homogeneity are Apache Spark[15] or Apache Flink[16], however ideally we would be engine agnostic, meaning that we would plug any engine independently of our processing logic. Therefore, we have decided to use Apache Beam[17]. Apache Beam is similar to the aforementioned options in that it is an open-source framework for parallel, distributed data processing at scale. It contrasts in the fact that it does not come with an execution engine of its own, but instead plugs into other execution engines, such as Apache Spark, Apache Flink, or Google Cloud Dataflow. Apache Beam is a unified model for defining both batch and streaming data-parallel processing pipelines, with a single API for both modes. This grants ample flexibility to share logic between each processing mode. As for the cloud deployment, there is Google Cloud Dataflow, a fully managed service for executing Apache Beam pipelines, fulfilling our needs. Let us revise the requirements for this component:

- *Scalability & Elasticity* - Both vertical and horizontal autoscaling work seamlessly in Dataflow. Dataflow dynamically adjusts the compute capacity allocated to each worker based on the utilization (vertical autoscaling). Dataflow estimates the appropriate number of worker instances required to run the processing job, and dynamically provisions

[15] https://spark.apache.org/
[16] https://flink.apache.org/
[17] https://beam.apache.org/

additional or fewer workers during runtime (horizontal autoscaling);

- *Reliability* - In case of worker failures, Dataflow may repeatedly retry pipeline code execution due to its built-in fault-tolerance support. It is able to ensure a level of fault-tolerance through the creation of backup copies of the pipeline code;
- *Exactly-once processing semantics* - Dataflow is able to provide *exactly-once processing semantics*, however with the use of non-deterministic sources or sinks, these semantics are not preserved[18]. Both Google Pub/Sub and the serving layer database are non-deterministic source and sink, respectively. Thus, further modifications will need to be performed to guarantee *exactly-once processing semantics*, which we will detail in Section 3.3.2.

Serving Layer

The serving layer is composed of a single primitive component: the database. It is accountable for persisting the aggregate intermediate results computed by the stream layer and serve query requests related to the stable and volatile signal sets. Both queries make use of a SUM aggregate function over the intermediate results network observations count, and either filter the resultant top 10% or bottom 10% observed networks, for the stable and volatile set queries, respectively. The database engine query language should allow us to express all of this query logic directly through it. To guarantee proper inter-layer operability and connectivity the database should be easily integrated with both the data processing stream layer component (Google Cloud Dataflow) and the API domain layer component (Kubernetes pod with the Java REST server), detailed in Section 3.3.1. *Writes* are expected to be made in as soon as possible, so the database component must support streaming records to it, and *reads* may be performed randomly. Additionally, the database must scale as the size of the intermediate results increases and be fault-tolerant. Based on these requirements, the most suitable cloud service candidates are Google Bigtable (Key-Value - NoSQL) and Google BigQuery (Relational - SQL). Both services are fully managed with scalability, high availability and fault-tolerance ensured. We decided to utilize Google BigQuery mainly due to its support of ANSI-standard SQL, granting us a higher level of expressiveness, and the seamless Google Dataflow integration for streaming records through the *Storage Write API*[19] with *exactly-once semantics*. Let us now go through the fulfilled criteria for this component:

- *Scalability* - BigQuery ensures system scalability as the dataset size increases from bytes up to petabytes. It is claimed that it maintains identical performance levels with minimal

[18] https://cloud.google.com/blog/products/data-analytics/after-lambda-exactly-once-processing-in-google-cloud-dataflow-part-1

[19] https://cloud.google.com/bigquery/docs/write-api

overhead;

- *Elasticity* - As a fully managed service, BigQuery ensures automatic resource provisioning. A sizable number of multi-tenant resources is kept pre-deployed in the background to rapidly scale up;
- *Reliability* - BigQuery storage attains fault-tolerance by automatically replicating storage across multiple locations ensuring a high level of availability and storage durability;
- *Standard SQL* - BigQuery supports ANSI-standard SQL (SQL:2011 [ISO11]) accommodating the previously mentioned aggregate functions and percentile computation necessary for the stable and volatile set query logic;
- *CROSS API Server integration* - BigQuery offers JDBC (Java Database Connectivity) drivers so that the CROSS API Server may interact seamlessly with its engine;
- *Dataflow integration for exactly-once delivery semantics* - Two methods are provided to insert data into BigQuery, either ensuring *exactly-once semantics* or a lower latency and potentially cheaper method with *at-least-once semantics*. We go into further detail on how *exactly-once semantics* are guaranteed in Section 3.3.2.

3.3.2 Supplemental Developments

In this section, we detail the several configurations that had to be performed, on top of the cloud deployment services chosen, to thoroughly satisfy the set requirements described in Section 3.1. Figure 3.8 details the deployed cloud architecture on the Google Cloud Platform.

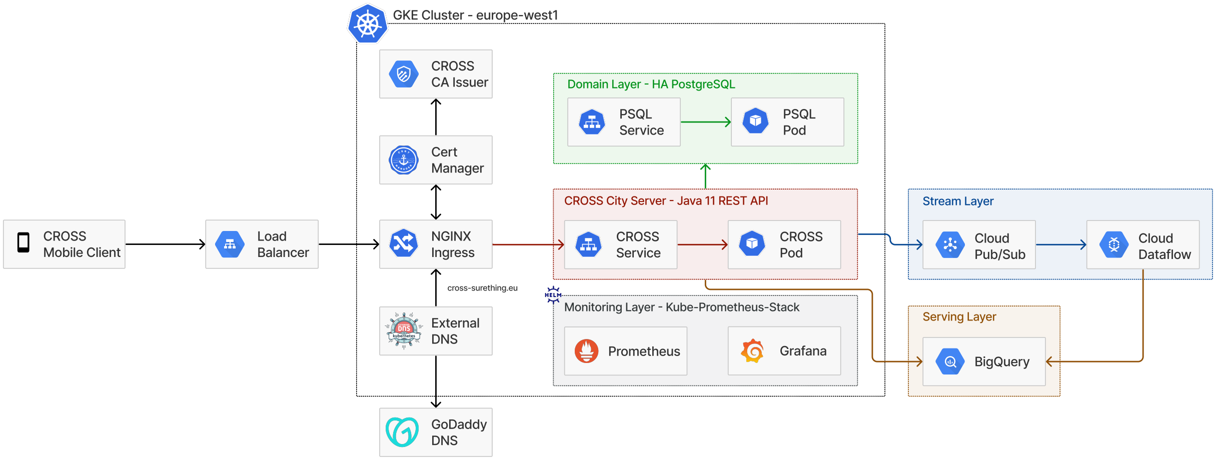

Figure 3.8: Overview of the CROSS City Cloud Google Cloud Platform Architecture.

Exactly-Once Processing Semantics

Most stream processing systems provide *at-least-once* guarantees, ensuring that records are always processed *at least once*. Cloud Dataflow uses the *upstream backup* technique to tolerate worker, network or other unexpected failures, assuring that each element is processed *at-least-once* across the various transformations. Meaning that the sender will attempt the element delivery again until it obtains a confirmation of receipt. Additionally, Cloud Dataflow ensures that it will keep retrying even if the sender crashes, guaranteeing that every record is delivered *at least once*. The issue, though, is that these retries could also produce duplicates, impeding *exactly-once processing semantics*.

The immediate solution for the record duplication problem is to tag every record sent with a *unique identifier*, and store a catalog of all identifiers that have already been seen and processed at each receiver. However, custom pipeline code running in Apache Beam can produce non-deterministic output. For example, a transformation can execute twice on the same input record, because of a retry, and yield a different output on each retry. Cloud Dataflow tackles this issue by employing *checkpointing* to effectively make non-deterministic processing deterministic. Before being delivered to the next stage, each output of a transform is saved to stable storage with its unique id.

A new problem arises when the pipeline is required to contact external remote services, as it typically happens at sources and sinks. Cloud Dataflow might retry reads from a source if processing fails, and needs to ensure that every unique record produced by a source is processed exactly once. For deterministic sources, Cloud Dataflow is able to deduplicate transparently. However, in our specific case, the source of our pipeline (Google Cloud Pub/Sub) is non-deterministic, due to its per-message parallelism where multiple subscribers can pull from a Cloud Pub/Sub topic, and the unpredictability of knowing which subscribers receive which message. In a processing failure, Cloud Pub/Sub will redeliver messages, however these may be delivered to workers other than those who handled them originally, and in a different order. To address this issue, we add a deterministic custom ID, equal to the hash of the concatenation of all the network observation fields, before publishing a network observation, as an additional message attribute. Dataflow then deduplicates messages, through an incremental aggregation mechanism, with respect to this deterministic message attribute ID. As a result, all processing logic can assume that the messages are already unique with respect to this custom ID. In short, we add an application-level deterministic identifier to each message, to allow each message to be distinguished and executed only once.

As previously mentioned delivering data externally in a sink is a side effect, and due to output non-determinism, exactly-once semantics are not transparently guaranteed. Thus, the least effort solution to guarantee that outputs are delivered exactly once in a sink, is to use the built-in sinks are provided as part of the Beam SDK. These sinks are carefully designed to prevent duplicate data from being produced, even when executed several times. Particularly in our case, Beam provides a BigQueryIO connector sink with Dataflow support to the BigQuery Storage Write API which is a unified data-ingestion interface that supports exactly-once semantics through the use of stream offsets. The Storage Write API ensure that it never writes two messages that have the same offset within a stream.

Load Balancing

The Kubernetes *Ingress*[20] resource manages external access to the services in our Google Kubernetes Engine (GKE) cluster. The Ingress provides load balancing, content-based routing (both host-based and path-based), and TLS termination. The Ingress exposes two HTTPS routes from outside the cluster to both the CROSS API and monitoring services within the cluster. An Ingress Controller must be deployed to the cluster, to satisfy the configuration defined in the Ingress resource. The Ingress Controller is, therefore, responsible for configuring an HTTP load balancer in Google Cloud, according to the Ingress resource. From the available Ingress Controllers, we have decided to utilize the *NGINX Ingress* Controller[21], as we had previous knowledge from former projects and it fulfills our set criteria.

With regards to domain name resolution, since the SureThing project already possessed domains purchased in the GoDaddy[22] DNS provider, we deployed an *ExternalDNS*[23] Controller, to the GKE cluster, to synchronize the exposed Kubernetes Ingress with the GoDaddy DNS provider. The ExternalDNS allows us to control DNS records dynamically in a DNS provider-agnostic way. With the use of this resource, we avoid the cost of maintaining a static IP address reserved to the CROSS City Cloud GKE cluster, since ExternalDNS automatically updates the DNS records in GoDaddy whenever we decide to provision a new cluster with a new IP.

Elasticity

Kubernetes *Deployments*[24] or *Stateful Sets*[25] controllers manage replicated applications and periodically guarantee that the current state matches the desired state. Nonetheless, this desired state of replica pods is typically static and set *a priori* in the workload configuration Kubernetes

[20] https://kubernetes.io/docs/concepts/services-networking/ingress/
[21] https://docs.nginx.com/nginx-ingress-controller
[22] https://www.godaddy.com/
[23] https://github.com/kubernetes-sigs/external-dns
[24] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[25] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

manifests. Kubernetes Deployments or Stateful Sets are not transparently elastic, and lack the ability to adapt to workload changes by deprovisioning or provisioning resources autonomously.

To attain elasticity in Kubernetes, we can leverage the *HorizontalPodAutoscaler*[26] resource to automatically update a workload resource, such as a Deployment or StatefulSet, with the goal of automatic horizontal scaling to adapt to the current demand. Horizontal scaling means that the response to increased load is to deploy more pods. If the load decreases, and the number of Pods is greater than the set minimum, the HorizontalPodAutoscaler instructs the workload resource to scale back down. Based on measured metrics such as average CPU utilization, average memory utilization, or any other custom metric, the horizontal pod autoscaling controller, operating within the Kubernetes control plane, periodically modifies the intended scale.

The metric used to keep track of effective utilization of the resources, during the application execution, is of extreme importance as it will dictate the desired scale as a response to the increase in load. Ideally, the metric used and threshold would be obtained through the analysis of each resource utilization during particular client workloads of former executions where the system was not able to meet the demand. However, in absence of such data we must resort to expected system and user behavior, which we go into further detail in the Section 4.5. As a specific example, a CROSS API replicated configuration could be set to horizontally autoscale based on an average CPU utilization threshold (*40%*), using the Kubernetes *Horizontal Pod Autoscaler* resource, provisioning identical CROSS API pods (*replicas*) to accommodate a growing work demand.

Monitoring Layer

An increased level of observability and analysis capabilities over each service of CROSS City Cloud, must be assured to aid system operators achieve high operational *maintainability.* Observability refers to the degree to which a system can be understood from its external outputs, such as CPU and memory utilization, disk space, latency, etc. Analysis refers to the activity of inspecting each observable data and retrieving useful information from it. To achieve this set goal, an additional monitoring layer/stack comprised of Prometheus[27] and Grafana[28] was added to CROSS City Cloud. The main responsibility of the monitoring layer is to collect, aggregate, and analyze metrics to allow increased understanding of the system behavior by its operators. Metrics are a measurement of a system at a given point in time that are intended to provide a picture of the system's health.

[26] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[27] https://prometheus.io/

[28] https://grafana.com/

Prometheus[27] is an open source, metrics-based monitoring system. Specifically in our monitoring layer, Prometheus is employed to collect and store real-time metrics as time series data, meaning that the metrics information is stored with the timestamp at which it was recorded. Kubernetes and Docker are already instrumented with Prometheus client libraries, exposing metrics in Prometheus format. *Exporters* may also be setup to expose metrics in different formats.

Grafana[28] is an open source analytics and interactive visualization web application, which allows us to query data stored in a Prometheus data source. The queries are made in PromQL (Prometheus Query Language) with built-in time-related query functionalities to leverage the time-series stored data. Grafana dashboards composed of charts and graphs can then be built, using our set of queries, to visualize relevant data.

The deployment of this layer to our GKE cluster was achieved through the use of the "kube-prometheus-stack" helm chart[29]. Helm assists the management of Kubernetes applications through *helm charts*, a collection of definitions that describe a related set of Kubernetes resources, meant to aid the deployment of significantly more complex Kubernetes applications.

Security Considerations

Due to the fact that CROSS City Cloud was deployed to the Google Cloud, a public provider, it is crucial to assure secure network communications between clients and the server. We should also provide a method for clients to authenticate the server. The usage of HTTPS (HTTP over TLS) protects the privacy and integrity of data transmitted while it is in transit. HTTPS authentication necessitates the use of a trustworthy third party to sign server-side digital certificates, hence we have setup a CROSS City project private Certificate Authority (CA) responsible for issuing certificates.

With regards to the Kubernetes deployment of our private CA, we have decided to utilize "cert-manager"[30] a X.509 certificate controller for Kubernetes and OpenShift workloads with the purpose of handling the certificate management. The CROSS project private CA is represented as an Issuer Kubernetes resource[31], able to generate signed certificates by fulfilling certificate signing requests. The Ingress resource, formerly described in Section 3.3.2, is secured through an additional request step for TLS signed certificates with additionally configured manifest annotations, during the its deployment. Moreover, cert-manager supports secure internal cluster service to service communication with mutual TLS, through the utilization of a Container Stor-

[29] https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

[30] https://cert-manager.io/

[31] https://cert-manager.io/docs/concepts/issuer/

age Interface Driver. This driver guarantees that the private key and associated signed certificate are unique to each pod and are saved on disk to the node of the cluster to which the pod is scheduled.

Another security concern is related to the access control to the GKE cluster Kubernetes resources. Only authorized system operator entities should be allowed to access the cluster resources. In our GKE cluster, tenant applications are assigned a specific Namespace, restricting them only to this Namespace. With this namespace model, we are able to leverage Role-based access control (RBAC), to control access to a particular namespace, for example to the *cross* namespace. The Kubernetes resource *Role*[32] defines the rules that represent the set of permissions allowed within the set namespace. This Role is then bound to specific system operators, represented by their Google Cloud accounts, through the Kubernetes resource *RoleBinding*[33]. On top of Kubernetes RBAC, used for controlling access to specific resources within the cluster, Google Cloud's Identity and Access Management[34] (IAM) was also be employed to control access to the cluster at the level of the Google Cloud project, which we will detail in Appendix B.2.

3.4 Summary

In this Chapter, we detailed the architectural extensions needed to ingest, aggregate and integrate scavenged network observations in pre-computed stable and volatile networks of given points of interest, both offline and in real-time. These extensions resulted in stream and serving layers, similarly to the Kappa architecture. The stream layer handles the asynchronous communication between the CROSS REST API server and the pipeline that produces the views of the aggregated network observations. The serving layer indexes the views and serves stable and volatile set query requests. Location claims made by tourists are then attested for the visited location and time of visit, through stable and volatile set queries. We presented the cloud deployment plan comprised of the cloud offerings and service abstractions for each system component. The deployment of the system leverages modern virtualization technology and the existing services offered by the Google Cloud Platform.

[32] https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding

[33] https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding

[34] https://cloud.google.com/iam

Chapter 4

Evaluation

In this Chapter, we describe the evaluation of the CROSS City Cloud solution.

In Section 4.1, we describe an assessment to determine compliance with each functional requirement outlined in Section 3.1. In Section 4.2, we describe the test dataset used throughout the quantitative evaluation, detailing both its collection process, as well as how we used it. In Section 4.3, we assess the feasibility in providing location and time-bound proofs. In Section 4.4, we present a systematic service characterization of CROSS City Cloud as an assistance to determine the quantitative evaluation goals, test workloads and metrics. In Section 4.5, we present and discuss the performance and scalability evaluation made to the domain layer. In Section 4.6, we showcase and analyze the stream layer performance and completeness evaluation. In Section 4.7, we present and discuss the assessment conducted to determine possible gains or losses observed in the serving layer, between our practical implementation and the implementation of the theoretical model. We conclude this Chapter with Section 4.8, where we summarize the experimental findings.

4.1 Qualitative Evaluation

The implemented CROSS City Cloud solution is able to fulfill each of the functional requirements enumerated in Section 3.1:

- **R1**: *Rewards should only be given to people who are actually eligible to receive them.* Each touristic trip setup by the system operators is composed of a series of points-of-interest with each having a pre-configured confidence threshold. A user's trip submission will only be accepted if it matches or surpasses the set confidence threshold of each claimed point of interest. The confidence is calculated based on the time of visit and the amount of network observation matches with the stable and volatile set, for time-unbound and time-bound locations proofs, respectively. Thus, rewards are only given to provers whose completed trip submissions are eligible;

- **R2**: *System operators should be able to identify which users were rewarded with which rewards.* Similarly to the CROSS V1 prototype, user reward history is persisted in the domain layer PostgreSQL database, as part of the user related information. Authorized system operators are able to fetch the user reward information by querying the domain database module, and thus obtain this information for further verification;

- **R3**: *Determine the stable network set by point of interest sliced by time.* Intermediate stable network sets are computed through the aggregation of user published network observations per point-of-interest and *period*. Stable network sets are meant to be computed within an *epoch*. An *epoch* encompasses a *period*, thus an *epoch* may vary from a *period* $\rightarrow \infty$. The stable network set of each individual point of interest can be queried with a minimum *period* time window granularity;

- **R4**: *Determine the volatile network set by point of interest sliced by time.* Intermediate volatile network sets are computed through the aggregation of user published network observations per point-of-interest in time windows of greatest common divisor of the *span* intervals set. Volatile network sets are meant to be computed within a *span*, and each of the possible spans encompasses smaller intervals of greatest common divisor size. The volatile network set of each individual point of interest can be queried with a greatest common divisor of the *span* intervals level of accuracy.

4.2 Lisbon Hotspots Dataset

The *Lisbon Hotspots* dataset (*LXspots*) [CEP22] was collected over a six month period across six different tourism locations distributed among the city of Lisbon, Portugal - Jerónimos, Comércio, Sé, Oceanário, Alvalade and Gulbenkian. The collected data is composed of discrete measurements of existing Wi-Fi networks. The measurements contain detailed information obtained through Wi-Fi scanning, such as MAC addresses and signal intensities. For redundancy, the data collection was done using three different smartphones running the Android operating system, namely: one Samsung Galaxy S9, one Huawei Mate 10 and one LG V10 thinq.

Due to its real-world relevance, the dataset is utilized as the base for the synthetic test workloads used in the subsequent evaluation sections, either by being the foundation of the expected user flow and access patterns or as the input.

4.3 Stable and Volatile Set Match as Location and Time Proof

We now assess the feasibility of our solution in providing location and time-bound proofs. We used the real-world collected network observations of the *LXspots* dataset, detailed in the previous Section 4.2. Each point-of-interest has different characteristics, such as being outdoors or indoors, sparse or central, and central or remote, hence differing proof effectiveness across each location is expected. Each smartphone - Samsung Galaxy S9, Huawei Mate 10 and LG V10 thinq - represents a distinct prover - Alice, Bob and Charlie. Each prover stays at each point-of-interest for 15 minutes. We will consider the seven consecutive day *epoch* from 2019-07-29 to 2019-08-04 and the one day *period* of 2019-08-19, available in the dataset.

4.3.1 Stable Set Match as Location Proof

To prove the presence at the location, the prover's collected network observation set, at the claimed point-of-interest, is compared against the set of stable networks computed for the previous *epoch*. Table 4.1 presents the percentage of match between these two sets. Considering a *50%* match threshold to determine successful proof, all provers visits are attested at four out of the six locations (except for Alice in Sé). Given the total 18 visits, this equates to a stable set success rate of 61.11%. Stable sets produced through our solution seem viable to attest presence at a location. Locations lacking stable networks such as Jerónimos and Comércio, due to their characteristics, would favour from either a lower match threshold or the deployment of known access points with TOTP, the dynamically changing SSID strategy described in Section 2.1.2.

4.3.2 Volatile Set Match as Time Proof

To prove the visiting period, the prover's scavenged network observation set, at the claimed point-of-interest, is compared against the set of volatile networks computed for the *span*. The match percentage between these two sets is shown in Table 4.1. The 15 minute visit is split into four span intervals - 15, 5, 3 and 1 min. Considering a *50%* match threshold to determine successful proof, visiting period attestation was achieved for all locations in at least 50.00% of the provers' claimed span intervals. Most notably, 75.00% of the claimed intervals in Comércio and Sé were successfully attested. Given the total 72 claimed intervals, this equates to a volatile set success rate of 63.89%. Our solution's volatile sets seem to be effective in attesting for the visiting period. It is also important to note, that longer intervals have a lower success rate than shorter intervals - 15 min (44.44%) and 1 min (83.33%). Due to the user's network scan collection period of 30 seconds, we can view a longer interval's volatile set as a union of several shorter intervals' volatile sets. Additionally, since a volatile set is composed of the bottom 10%

observed networks, if a specific subset of shorter intervals was ”more observed” than the other, then the most observed subset will have a greater influence on their union. Meaning the volatile set of the longer interval will be different from its subset shorter intervals. This demonstrate a higher dependence on consistent colocated witnesses for long visiting period proof, since ideally network observations should be scavenged equally, meaning by the same number of witnesses, throughout the full duration of the interval. Nonetheless, touristic visits are typically performed in specific groups and scheduled intervals, thus the feasibility of the solution remains plausible.

Table 4.1: Prover’s Stable and Volatile Set Match Percentage for each Point-of-Interest (percentage $\geq 50\%$ shown in green, and $< 50\%$ in red).

Point-of-Interest	Prover	Stable Set Match	Stable Set Success Rate (≥ 50.00%)	Volatile Set Match for Claimed Span Interval				Volatile Set Success Rate (≥ 50.00%)
				15 min	5 min	3 min	1 min	
Alvalade	Alice	100.00%	61.11%	100.00%	87.50%	100.00%	90.00%	63.89%
	Bob	100.00%		0.00%	61.53%	50.00%	58.33%	
	Charlie	92.85%		0.00%	30.76%	62.50%	46.15%	
Comércio	Alice	27.77%		20.00%	50.00%	0.00%	100.00%	
	Bob	30.55%		57.14%	100.00%	100.00%	100.00%	
	Charlie	27.77%		80.00%	0.00%	100.00%	100.00%	
Gulbenkian	Alice	100.00%		0.00%	12.50%	50.00%	91.66%	
	Bob	60.70%		54.54%	41.66%	33.33%	46.15%	
	Charlie	100.00%		44.44%	50.00%	78.57%	83.33%	
Jerónimos	Alice	9.30%		30.00%	50.00%	60.00%	75.00%	
	Bob	27.90%		9.09%	30.00%	25.00%	40.00%	
	Charlie	20.93%		54.54%	50.00%	33.33%	100.00%	
Oceanário	Alice	85.00%		83.33%	100.00%	100.00%	100.00%	
	Bob	65.00%		0.00%	50.00%	50.00%	60.00%	
	Charlie	75.00%		16.66%	14.28%	14.28%	50.00%	
Sé	Alice	43.00%		60.00%	86.00%	33.33%	100.00%	
	Bob	50.00%		62.50%	66.60%	50.00%	75.00%	
	Charlie	54.00%		0.00%	33.33%	50.00%	75.00%	

4.4 Systematic Service Characterization

The subsequent sections of the evaluation will focus on the *performance* and *scalability* of the resulting system. The process will involve the collection of key metrics to quantify the demands, performance of the system and assess the resource efficiency during the execution of demanding synthetic workloads. Each layer is evaluated separately, as each impacts the performance differently. We start by doing a systematic service characterization of each layer containing a description of the service it provides, possible high level workloads and important metrics to collect, illustrated in Table 4.2. The baseline values for each metric are obtained through the deployment of the system with a basic configuration and detailed in each section. During the execution of the synthetic workloads, it is also important to assess the utilization and efficiency of the underlying resources more specifically the CPU and memory.

Table 4.2: CROSS City Cloud service characterization and benchmark model.

	Domain Layer	**Stream Layer**	**Serving Layer**

Services	Authenticate users;
	Serve domain touristic and user information requests;
	Verify location proofs;
	Assign rewards.
		Store network observations;
		Process network observations to produce intermediate aggregate results.
			Fetch stable and volatile network set data.

Workloads	Series of read and write type of domain data requests; Series of distinct trip submissions.	Series of observation submissions of different POI and collection timestamp.	Series of stable and volatile network set requests of different POI and time granularity.

Metrics	Request response time;
	CPU Usage;
	Memory Usage.
		Rate of observations processed;
		Observation process latency;
		Data watermark lag;
		CPU Usage;
		Memory Usage.
			Response time;
			Slot time.

Ideally the test workloads used would be obtained through traces of real execution of the system, however since this is a brand new development we must resort to expected user behaviour.

4.5 Domain Layer Scalability and Performance Testing

The domain layer is comprised of the API and database deployed to a cloud environment, as detailed in Section 3.2. To assess the performance and scalability of this layer, we synthesized a test workload based on expected user access patterns. We focus on the *submission of trip visits* through the domain layer as this is the typical path of a user's interaction with CROSS. The integration of scavenged network observations is done asynchronously, through the stream and serving layer, and will be evaluated separately. Ideally the test workload used would be obtained through traces of real execution of the system, however since this is a new system we resort to predicted user scenarios. With the execution of the test workload we stress each component of the SUT (System under Test). During the execution of the workload, the request response time, rate of requests and both the CPU and memory usage metrics were collected to assess the user perception of the system and adequate resource utilization.

Several benchmark load testing tools were considered for executing the test workload, most notably k6[1], wrk[2], and Apache JMeter[3]. k6 was chosen as the benchmark load testing tool. The

[1] https://k6.io/

[2] https://github.com/wg/wrk

[3] https://jmeter.apache.org/

main grounds sustaining the k6 tool decision are the *"Everything as code"* doctrine with test logic and configuration options written in JavaScript for both ease of version control (important for reproducible testing) and integration with our Protocol Buffer payloads. The tool itself is written in Go to achieve maximum performance, embedding a JavaScript runtime.

4.5.1 Scalability Model

Before delving into the experimental results and discussion, we briefly present our rationale for using a scalability model, and the model used in the experiment. We will be using a scalability model to assess the behavior of the system beyond the capacity used in the experiments and to quantify it in specific parameters which can be compared with. in the The *Universal Scalability Law* (USL) [Gun06] (Equation 4.1) extends *Amdahl's law* [Amd67] (Equation 4.2) with an additional parameter (κ), allowing us to model capacity degradation related to coherency losses. Other scalability models exist such as the Exponential, Geometric and Quadratic. The USL, on the other hand, differs from the other models in that it is defined in terms of two parameters rather than a single one, accounting separately for both contention (serial work) and coherence (crosstalk among workers in the system such as nodes, CPUs, threads, etc). The contention component (σ) of the system ends up limiting asymptotically its speedup, while the coherence portion (κ) limits the maximum system achievable size. Using a nonlinear regression, we are able to solve the equation and determine best-fit values for the three parameters λ, σ and κ. Nonetheless, we will be using the USL R package [Möd20] to compute the parameters, reducing the manual work needed to perform the scalability analysis.

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)} \tag{4.1}$$

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1)} \tag{4.2}$$

X = throughput

N = concurrent users

λ = performance coefficient

σ = serial portion

κ = crosstalk factor

4.5.2 Experimental Setup

The benchmark tool alongside the test scripts were wrapped up into a Docker image, tagged, and pushed to our Google Cloud project's container private registry. The k6 image *version 0.38.3* provided by Grafana Labs[4] was used as the base image, with the CROSS Certificate Authority added to the system's trusted certificates pool.

The benchmark tool and CROSS City Cloud's domain layer were deployed to distinct Google Kubernetes Engine (GKE) clusters, comprised of one and two nodes, respectively, physically isolated, in the "europe-west1 region". A comprehensive specification of the GKE clusters used is detailed in Table 4.3. It is worth noting that no resource limit was configured at any point to avoid resource contention when performing the tests.

Table 4.3: Google Kubernetes Engine cluster specification for CROSS City Cloud and the k6 benchmark tool.

Stack	**Machine Family**	**Machine Type**	**OS**	**Kernel Version**	**CPU**	**RAM Size**	**Disk Size**	**Disk Type**	**Additional Disk Size**	**Additional Disk Type**	**Network**	**Docker Ver**	**Kubernetes Ver**	**Region**	**Locations**
CROSS City Cloud	General-Purpose	e2-highcpu-4	Container-Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (4)	4 GB	20 GB	Standard persistent disk	-	-	Default	20.10.12	1.21.11-gke	europe-west1	europe-west1-b, europe-west1-c
k6 Benchmark Tool	General-Purpose	e2-highcpu-8	Container-Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (8)	8 GB	20 GB	Standard persistent disk	200 GB	Regional Standard Persistent Disk	Default	20.10.12	1.21.11-gke	europe-west1	europe-west1-b

The test workload was comprised of two stages of *distinct load duration* and *ramping user concurrency*. The first stage lasts for five minutes, while the second stage lasts for ten minutes, which constitutes the sum total of fifteen minute load duration. Each stage executes an identical user flow: the users sign-in to authenticate, retrieve existing routes, fetch a specific route, and submit a visit to one of the route's point of interest with a sufficient amount of Wi-Fi AP evidences to achieve the route's waypoint set confidence threshold (75%), claiming that location. It is worth noting that the route used in the test accepts multiple submissions for the same user, to ease testing, and that various user accounts are utilized. The first stage ramps from zero to ten concurrent virtual users (vus) and the second stage ramps from ten to fifty concurrent virtual users, as shown in Figure 4.1.

The test workload will be executed in three separate configurations - A, B and C - of the domain layer, with regards to the CROSS API server level of replication. The baseline configuration is comprised of a single replica (A), while the test configurations vary from one to two replicas (B) and from one to four replicas (C). Since the workload is expected to be CPU bound, each configuration horizontally auto-scales based on a set average CPU utilization threshold (*40%*), using the Kubernetes *Horizontal Pod Autoscaler* resource, which provisions

[4] https://hub.docker.com/r/grafana/k6/

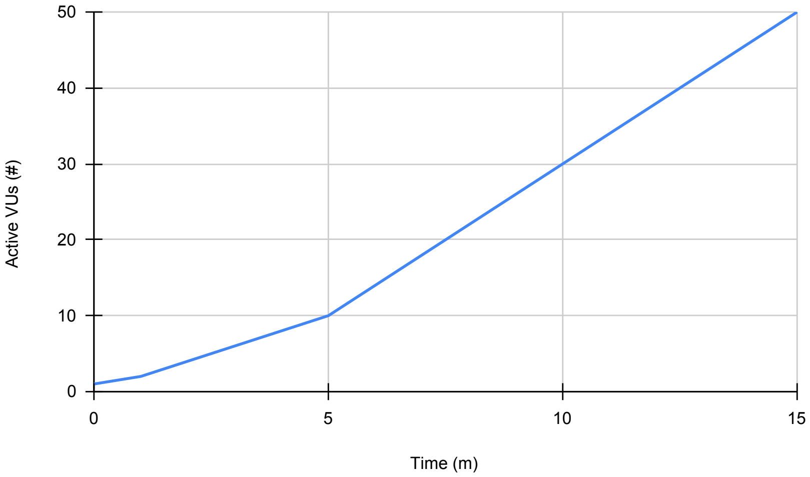

Figure 4.1: Concurrent Active Virtual Users over Time during the execution of the Test Workload.

identical CROSS API server pods (*replicas*) to accommodate a growing work demand.

4.5.3 Results

As previously stated, we have plotted the experimental results over four distinct system performance metrics, to assess system and user observed behaviour, and resource usage. Figures 4.5 to 4.2 consist of plots over the mean, median, percentile 90 and minimum latency, in milliseconds, and mean throughput, in requests per second, while Figures 4.9 to 4.13 show the mean CPU usage per service and per pod, in CPUs or cores, and the mean memory utilization per service and per pod, in MiB and GiB.

4.5.4 System Performance and Scalability Analysis

Using the throughput measurements collected per level of user concurrency, plotted in Figure 4.2, we are able to model the scalability of the system with the *Universal Scalability Law* (USL) (Equation 4.1). Table 4.4 summarizes the performance coefficient and scalability parameters estimations, detailed in Section 4.5.1, for each configuration, with their respective model plots in Figure 4.3.

Table 4.4: *Universal Scalability Law* (USL) parameters for each configuration.

Configuration	λ	σ	κ
A	23.24	0.0409	0.0007583
B	23.57	0.0000	0.0006959
C	26.87	0.0000	0.0003925

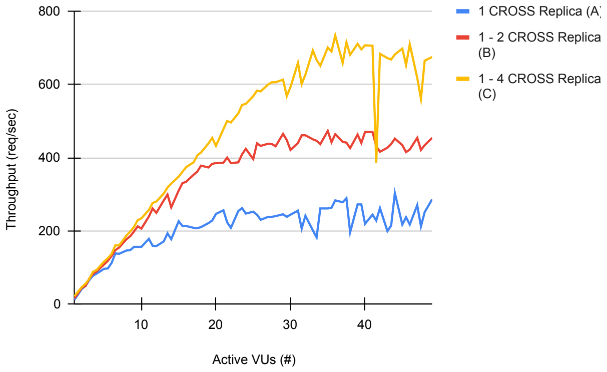

Figure 4.2: Mean Throughput over Concurrent Virtual Users.

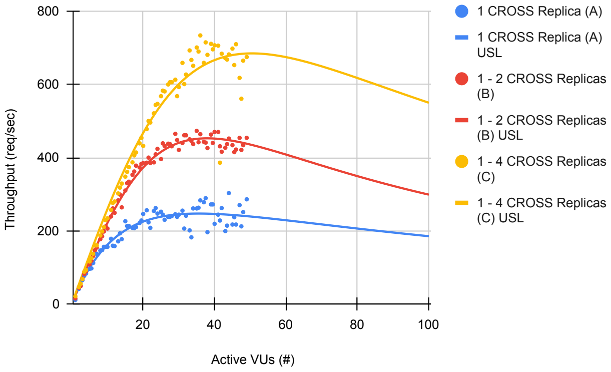

Figure 4.3: Throughput over Active Concurrent Virtual Users with USL model.

By comparing the estimated performance coefficients (λ) at the unitary load, we can quantify the efficiency of the system across sizes. Doubling the size of the system from both configuration A to B and B to C we maintain approximately 51% and 57% efficiency, respectively. Despite the efficiency values being lower than expected, these can be explained by the fact that all configurations start the test workload execution with the same amount of replicas (1), and only when above a certain average CPU utilization threshold do configurations B and C provision the additional resources to accommodate the extra demand. Regarding the scalability of the system, the maximum useful user concurrency of each configuration, calculated through the Equation 4.3 in relation to both scalability parameters, is 35, 37 and 50 users for A, B and C, respectively, after which point performance is significantly degraded. Based on the maximum useful user concurrency, the speedup achieved on the observed maximum throughput between configuration A (247 req/sec) and C (684 req/sec) is of approximately $2.77x$. From a request performance standpoint, we can model latency in relation to throughput by combining the *Universal Scalability Law* (Equation 4.1) and *Little's Law* [All90] (Equation 4.4) resulting in Equation 4.5. By observing Figure 4.4, which plots the *modelled* relation between latency and throughput for each configuration, we note both a higher achievable throughput, as previously discussed, and lower latency for configuration C, as well as the expected degradation at the aforementioned observed maximum throughputs.

$$N_{\max} = \frac{\sqrt{1-\sigma}}{k} \tag{4.3}$$

$$N = XR \tag{4.4}$$

$$R(X) = \frac{-\sqrt{X^2(\kappa^2 + 2\kappa(\sigma - 2) + \sigma^2) + 2\lambda X(\kappa - \sigma) + \lambda^2} + \kappa X + \lambda - \sigma X}{2\kappa X^2} \tag{4.5}$$

N_{max} = maximum achievable concurrent users

σ = serial portion

κ = crosstalk factor

N = concurrent users

X = throughput

R = response time

λ = performance coefficient

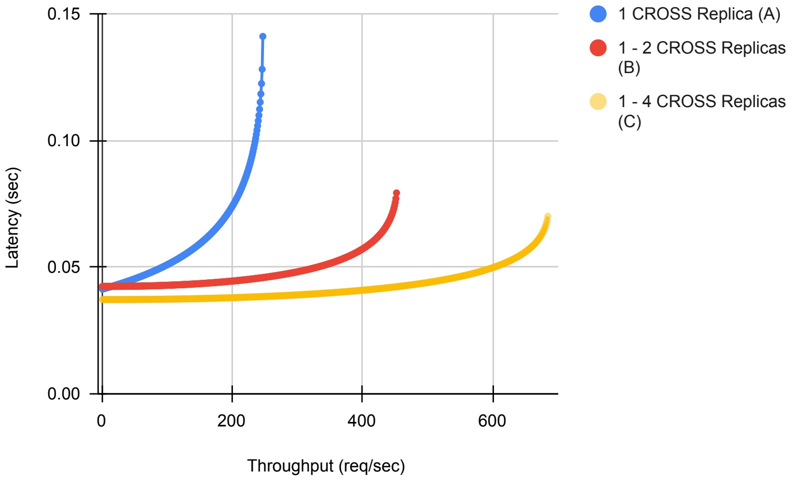

Figure 4.4: Latency modelled in relation to Throughput for each system size configuration.

4.5.5 Request Performance Analysis

From a practical standpoint, derived from the collected mean latency measurements plotted in Figure 4.5, we are able to note that for a set latency threshold of 100 ms (the limit for giving the user the perception that the system is reacting instantly [Mil68, CRM91]), configuration A is able to perform below set threshold solely until 17 concurrent virtual users after which a high level of degradation is noticeable. Configuration B, by leveraging the second node, is able to withstand set threshold until 37 concurrent virtual users after which a similar level of performance degradation is observed. Despite the peak at 41 concurrent virtual users, configuration C is capable of performing below set threshold for the full duration of the test workload. This peak is also observed in the median latency plotted in Figure 4.6, thus it is not an outlier skewing the average response time, and neither CPU or memory utilization (Figures 4.9 and

4.10) seem abnormal at the minute 13, hence why one can attribute this peak to either a low surge of requests from the client benchmark cluster side or noisy neighbours.

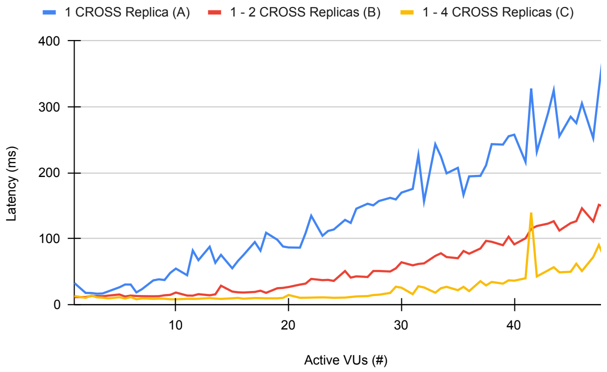

Figure 4.5: Mean Latency over Active Concurrent Virtual Users.

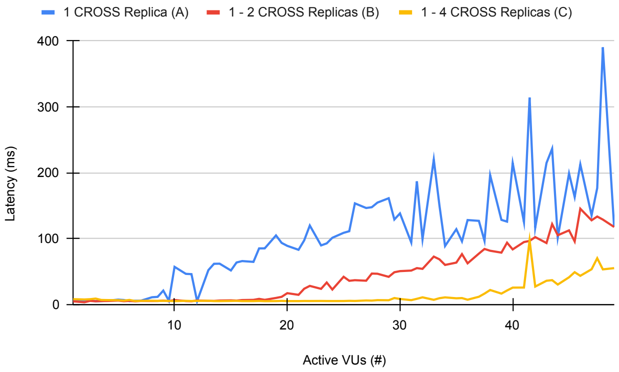

Figure 4.6: Median Latency over Active Concurrent Virtual Users.

Additionally on the subject of request performance from the practical standpoint, in an effort to further quantify the overall user experience and perception of the system with each configuration, we have plotted the percentile 90 latency (filters top 10% worse latencies) in Figure 4.7. Percentiles are useful for us to determine the expected maximum response time for a percentage of requests/users. In this particular case, only configuration C (in spite of the aforesaid peak at 41 concurrent virtual users) is able to able to withstand a set latency threshold of 200 ms (double the set mean latency threshold), meaning we are able to conclude that for 90% of users, within the tested range of user concurrency, will experience a response time either as fast or faster than 200ms. From the minimum latency plot in Figure 4.8, we are able to infer that both configuration B and C with similar mean minimum latencies of 2.10 and 2.30 ms, respectively, are consistently below configuration A, which demonstrates sensitive behavior to spikes from 24 concurrent users onwards (11 users below the predicted the maximum useful user concurrency of this configuration) due to contention.

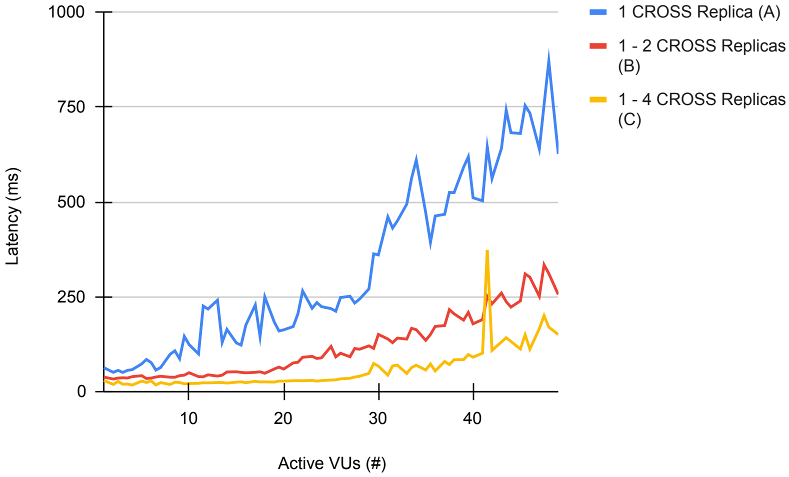

Figure 4.7: Percentile 90 Latency over Active Concurrent Virtual Users.

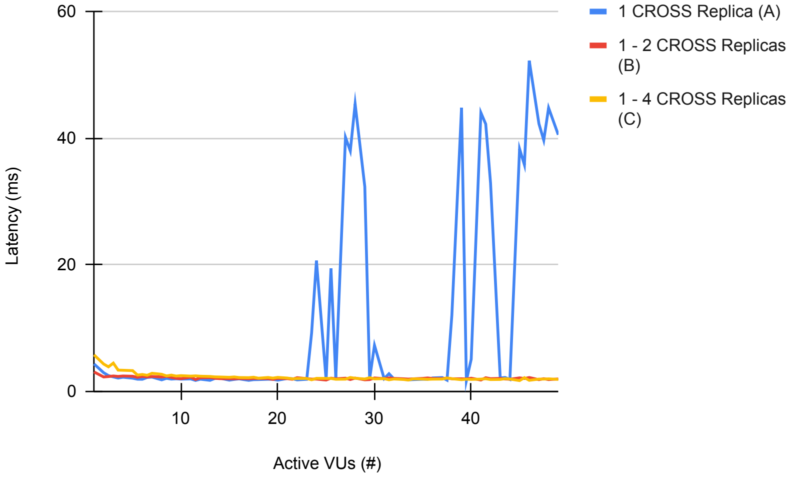

Figure 4.8: Minimum Latency over Active Concurrent Virtual Users.

4.5.6 Resource Utilization Analysis

Regarding resource utilization, from the CPU and memory usage per pod plots in Figures 4.11 and 4.13 we can infer that both configurations B and C provision the additional pods at minute 3 (cold start) and become operational between minute 4 and 5. Furthermore, from the PostgreSQL CPU usage plot in Figure 4.12 we are able to deduct that, for this particular user access pattern, higher levels of user concurrency have a higher impact on PostgreSQL than the CROSS API, and only between minute 6-7, and 11-12 do the configurations B and C, respectively, utilize more CPU from PostgreSQL than configuration A. As expected configuration C reaches a higher level of resource utilization on both CPU, overall CPU usage plot Figure 4.9, (A - 2.38 CPUs, B -

3.90 CPUs and C - 4.79 CPUs) and memory, overall memory usage plot Figure 4.10, (A - 2.06 GiB, B - 2.37 GiB and C - 3 GiB), albeit significantly within the cluster limits of 8 CPUs and 8 GiB.

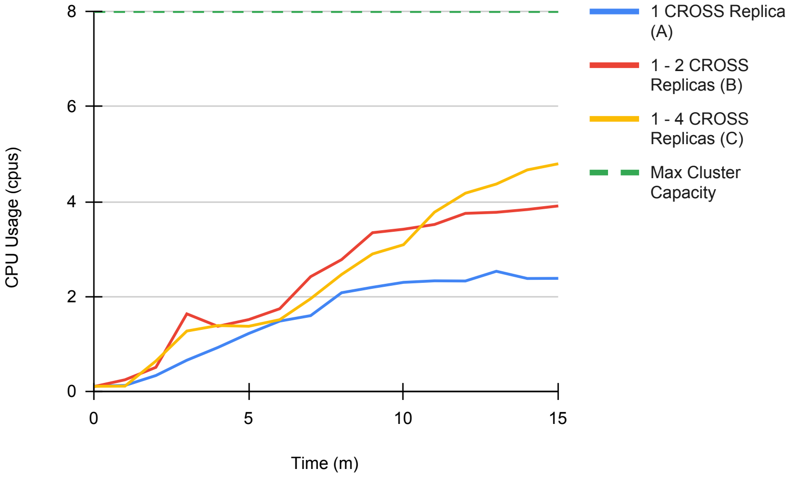

Figure 4.9: Mean CPU Usage over Elapsed Time.

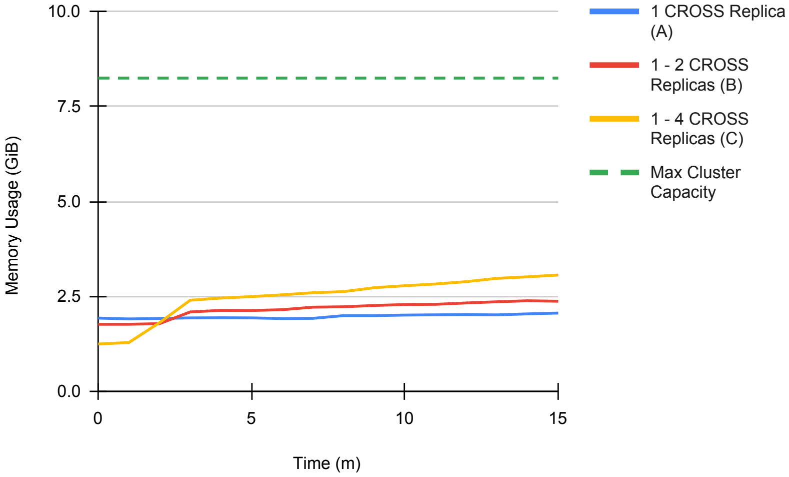

Figure 4.10: Mean Memory Usage (without cache) over Elapsed Time.

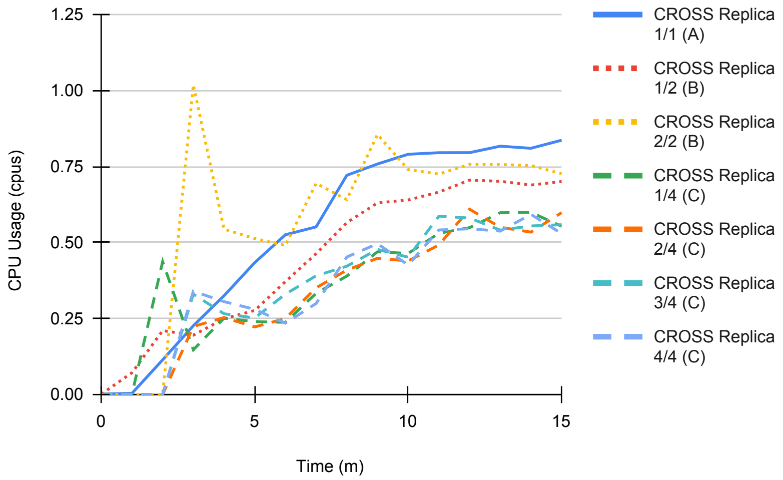

Figure 4.11: Mean CPU Usage (per pod) over Elapsed Time.

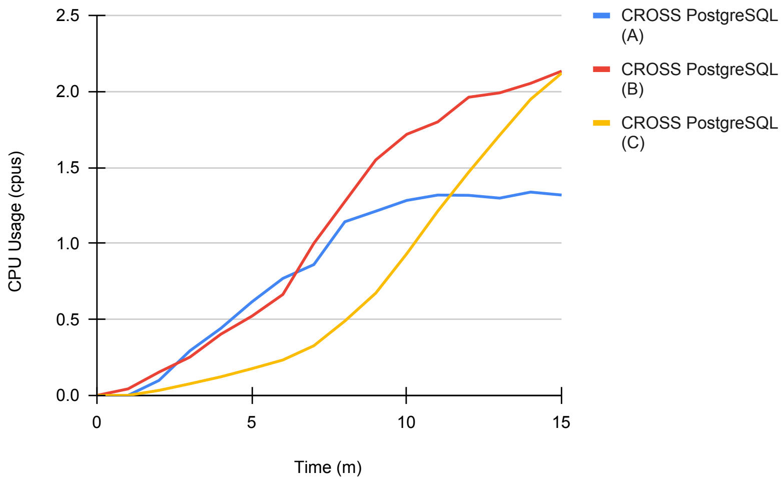

Figure 4.12: Mean PostgreSQL CPU Usage over Elapsed Time.

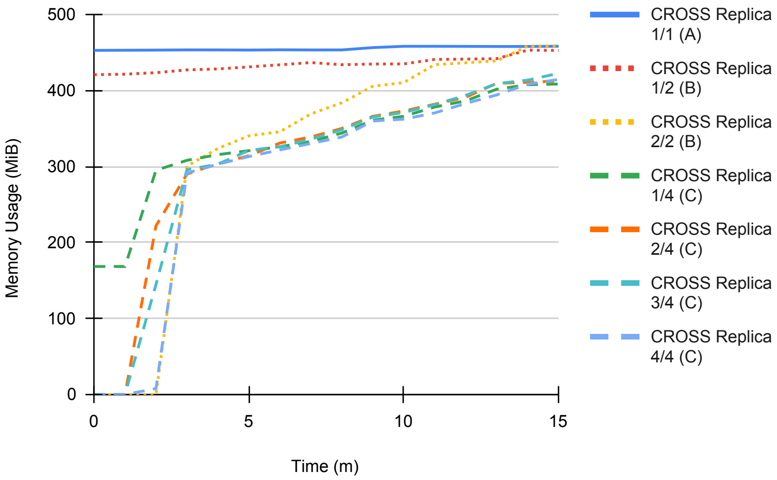

Figure 4.13: Mean Memory Usage (per pod, without cache) over Elapsed Time.

4.5.7 Summary

For this specific synthetic workload, configuration C is capable of outperforming the other two configurations with regards to both system and request performance metrics, as observed and forecasted through the scalability model. Moreover, we infer that the system is able to scale horizontally, while maintaining an acceptable level of performance and resource utilization.

4.6 Stream Layer Performance and Completeness Testing

The stream layer pipeline is implemented by Apache Beam and is responsible for processing the published network observations and producing intermediate aggregate results. A real-time scenario was setup, to estimate the trade-offs made in performance, completeness and cost of the pipeline. During the execution of the scenario test workloads, a set of metrics were collected including the rate of observations processed, the observation process latency which refers to the maximum time that a particular network observation has been processing or awaiting processing in the pipeline, data watermark lag. The data watermark lag refers to the amount of time since the most recent output watermark - network observation publish time. CPU and memory usage were also collected to ensure adequate resource usage with no limits, avoiding any possible contention when performing the tests.

The test workload was performed using a custom Java client developed to simulate real-time user behaviour of the network observation collection and submission process.

4.6.1 Experimental Setup

The custom Java client was wrapped up into a Docker image tagged and pushed to our Google Cloud project's container private registry, for ease of result reproduction used the Maven image *3.8-jdk-11*[5].

The client was deployed to a GKE cluster, comprised of a single node in the "europe-west1" region. A comprehensive specification of the GKE clusters used is detailed in Table 4.5. The Message Broker and the Apache Beam pipeline were deployed to Google Cloud Pub/Sub and Google Cloud Dataflow, respectively.

Table 4.5: Google Kubernetes Engine cluster specification for the real-time client.

Stack	Machine Family	Machine Type	OS	Kernel Version	CPU	RAM Size	Disk Size	Disk Type	Additional Disk Size	Additional Disk Type	Network	Docker Ver	Kubernetes Ver	Region	Locations
Real-Time Client	General--Purpose	e2--highcpu--8	Container--Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (8)	8 GB	20 GB	Standard persistent disk	200 GB	Regional Standard Persistent Disk	Default	20.10.12	1.21.11--gke	europe--west1	europe--west1-b

The real-time processing test workload consists of real time submissions of a dynamic set (both in size and frequency of observations) of network observations of a specific point of interest as Wi-Fi AP evidences, simulating both user collection and submission. The simulated point of interest is Sé due to its level of volatility, which is present in the test dataset described in Section 4.2. Each client submission contains a sample of collected/observed Access Points from the total number of existing Access Points in the point of interest. This synthetic sample is derived from the probability of occurrence of each Wi-Fi Access Point spotted in Sé during the days of 2019-07-29 and 2019-08-19, which is plotted in Figure 4.14.

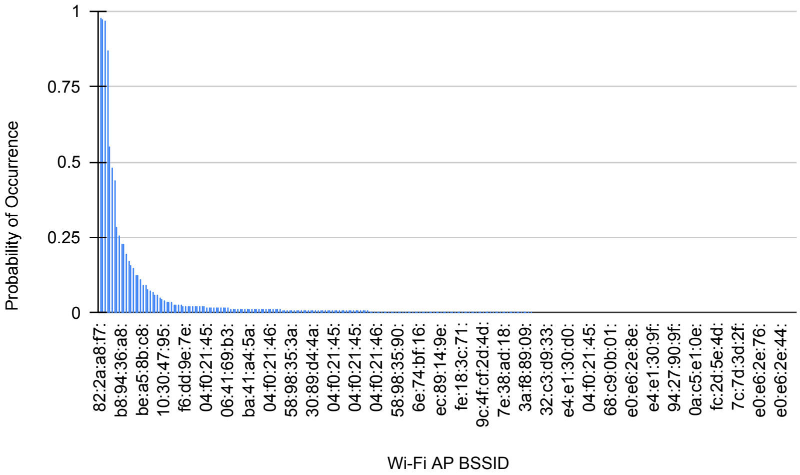

Figure 4.14: Observed probability of occurrence, in a sample, of each Wi-Fi Access Point spotted in Sé during 2019-07-29 and 2019-08-19.

[5] https://hub.docker.com/_/maven

As we can see, the distribution has a long tail, meaning that the sample publish time is skewed by a random amount of publish time delay (maximum 1 minute delay) from the current system clock time (*simulating late events*) and the sample has a certain probability of being published multiple times (*simulating duplicate events*). The test workload has a total duration of 20 minutes, to provide ample time to quantify both the performance and impact, as well as trigger multiple stable and volatile set intermediate windows. Regarding the time window parameters, the volatile window size is set to 1 minute, calculated as the greatest common divisor of the *spans* (1, 5, 10 and 15 minutes) which are derived from each respective *delta* (30 seconds, 2.5, 5 and 7.5 minutes). The *deltas* are set by the *system operator* entities and represent the several achievable levels of accuracy. We decided to use only the low value deltas, as they are the most useful, in contrast to the maximum possible *span* which is a *period* (1 day). The stable window size is typically set to a *period* (1 day), however since the test workload has a duration of 20 minutes we have set it to 5 minutes, which results in 4 stable window triggers, in an effort to observe the impact of the stable set pipeline stages.

4.6.2 Results

The results are grouped according to the pipeline stage and/or phase depicted in Figure 4.15, which illustrates the Dataflow implementation of the architected pipeline DAG from Figure 3.7. Figures 4.16 to 4.20 consist of plots over the *throughput*, in observations per second, while Figures 4.21 to 4.22 plot the mean *latency*, in seconds, and the *data watermark lag*, in minutes. Resource utilization (CPU and memory) is plotted in Figures 4.23 and 4.24. All of the aforementioned plots draw comparisons between the pipeline's processing semantics (*exactly-once* and *at-least-once*), which is the factor that is expected to have the most impact on the performance of this layer, based on the explanation detailed in Section 3.3.2. Therefore, through the analysis of the plots we can properly quantify this impact. Note as well that, as detailed in Section 3.2, Dataflow assures *at-least-once semantics* by default, however to guarantee *exactly-once semantics* modifications had to be performed to ensure that both our source (Pub/Sub - Read WiFi AP Obs Proto from Pub/Sub Stage) and sinks (BigQuery - Write Intermediate Results Stages to BigQuery) were deterministic. Thus, these stages are a good starting point for our analysis.

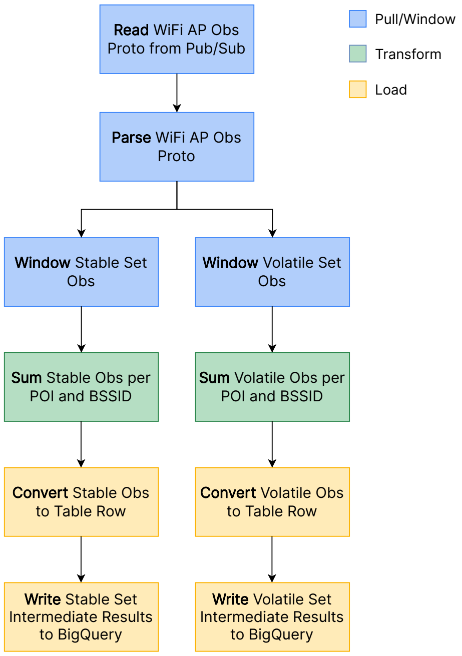
Read WiFi AP Obs Proto from Pub/Sub
Parse WiFi AP Obs Proto
Window Stable Set Obs
Window Volatile Set Obs
Sum Stable Obs per POI and BSSID
Sum Volatile Obs per POI and BSSID
Convert Stable Obs to Table Row
Convert Volatile Obs to Table Row
Write Stable Set Intermediate Results to BigQuery
Write Volatile Set Intermediate Results to BigQuery
Pull/Window
Transform
Load

Figure 4.15: CROSS Dataflow pipeline Directed Acyclic Graph (DAG).

4.6.3 Impact on Throughput

From the plots in Figures 4.16 and 4.17 of the overall throughput over the total test workload duration, we are able to observe that, for the same rate of observation publish requests, the pull/window pipeline phase containing the Pub/Sub source with *at-least-once semantics* is able to process more than 700 observations per second (Figure 4.16), where as with *exactly-once semantics* it is only able to process more than 200 observations per second (Figure 4.17), equating to a 3.5x speedup. However, both semantics seem to process observations at a similar rate in the transform and load phases, in spite of our contrary expectations.

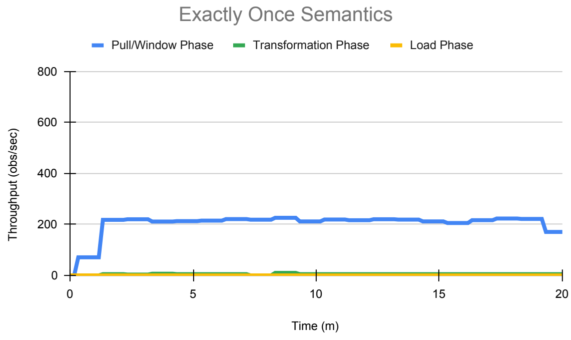

Figure 4.16: Throughput per phase of the pipeline with *exactly-once semantics.*

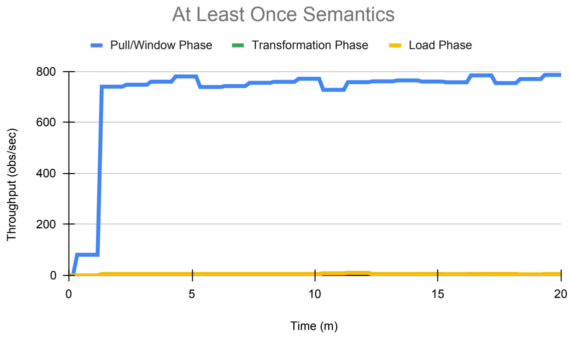

Figure 4.17: Throughput per phase of the pipeline with *at-least-once semantics.*

In more detail, we have plotted the mean throughput per pipeline stage in Figures 4.18 to 4.20. From the plot in Figure 4.18, we are able to confirm the speedup observed, of approximately 3.5x, with *at-least-once semantics* all throughout the pull/window phase in which each stage is able to performance consistently at the same rate. Nonetheless, as noted in the transform phase plot in Figure 4.19 the performance gains obtained in the pull/window phase with *at-least-once semantics* are lost, and both semantics end up performing at identical levels (approximately 1.0x speedup), indicating a potential bottleneck at these stages. In the load phase (Figure 4.20), with *at-least-once semantics* the pipeline is able to sustain the rate of observations from the previous phase, confirming the bottleneck suspicion with these semantics, as opposed to the pipeline with *exactly-once semantics* which is not able to maintain the rate at this phase, more specifically at the *Write to Big Query* stages where we observe a 165x speedup, demonstrating that these are in fact the bottleneck stages with these semantics.

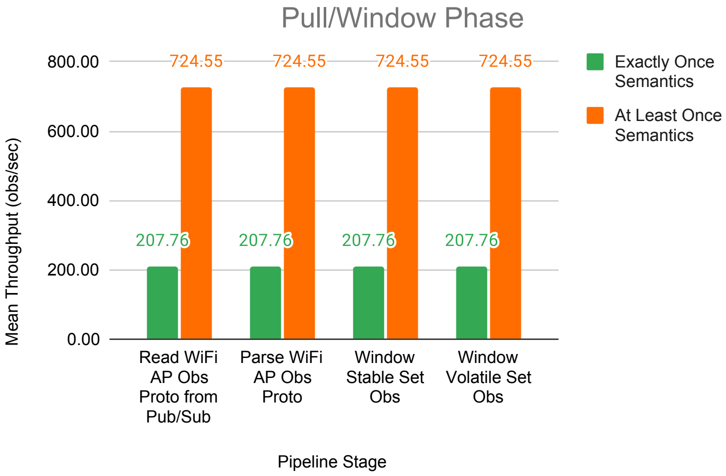

Figure 4.18: Mean throughput per pipeline stage in the pull/window phase for each semantics.

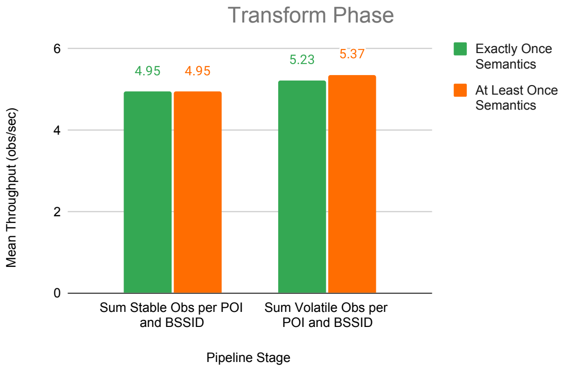

Figure 4.19: Mean throughput per pipeline stage in the transform phase for each semantics.

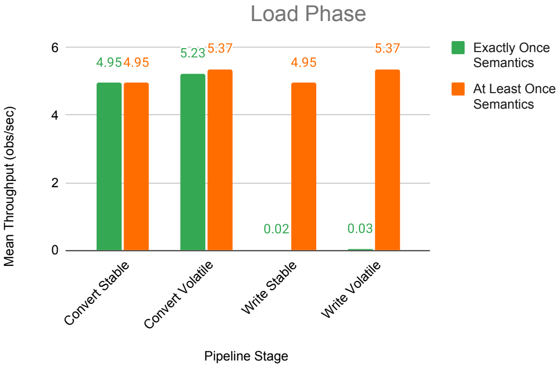

Figure 4.20: Mean throughput per pipeline stage in the load phase for each semantics.

4.6.4 Impact on Network Observation Processing Time

Regarding the time of processing, latency or system lag, from the perspective of a single observation in the pipeline, we have plotted the mean latency per pipeline phase, in seconds, in Figure 4.21. By observing the aforesaid plot, we conclude that in spite of the significant speedup in throughput over the pull/window phase with *at-least-once semantics*, this equates to a 1.1x speedup in latency. Additionally, the load phase impact is as expected noted in the transform + load phases, in which the *at-least-once semantics* pipeline has a 3.0x speedup over the *exactly-once semantics* pipeline. It is important to note however that this plot contains each phases cumulative latency, meaning that if Dataflow is able to parallelize these stages, the latency differences on the overall system can be negligible.

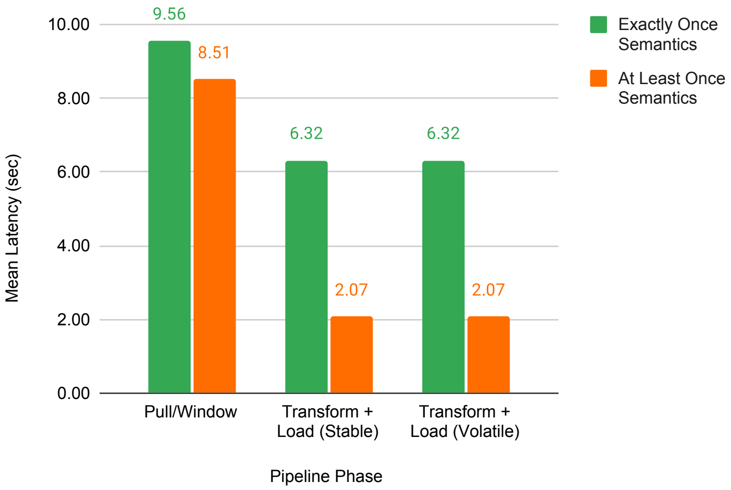

Figure 4.21: Mean latency per pipeline phase for each semantics.

4.6.5 Impact on Output Data Watermark

Another important metric that allows us to quantify the processing time in relation to the publish time of an observation (the client collection time), which in our pipeline is used as the observation's watermark, is the data watermark lag. The data watermark lag, as previously explained, is the age up to which all data has been processed by the pipeline, meaning that it is the time since the earliest watermark output by the pipeline at a specific point in time. In this specific test workload we know that observations can be delayed at most 1 minute, thus we expect that the data watermark lag to be at least 1 minute plus the cumulative processing time in the CROSS API server, Pub/Sub and the pipeline. We are able to observe that as expected both pipelines have a data watermark lag of at least 1 minute, and in spite of two peaks at 2 and 7 minutes in the *exactly-once semantics* pipeline, both are performing at a similar level, as plotted in Figure 4.22. More precisely, the pipeline with *at-least-once semantics* and the pipeline with *exactly-once semantics* achieve a mean data watermark lag of 1.35, 1.46, respectively, a median data watermark lag (which excludes the peak outliers) of 1.20 and 1.26, respectively. These differences total to a speedup of the *at-least-once semantics* pipeline over the *exactly-once semantics* pipeline of 1.08x and 1.05x for the mean and median, respectively.

Regardless of the processing semantics enforced in the pipeline, from these results we are able to derive the overall median system lag per observation from the complete flow throughout this layer (including the CROSS API server, Pub/Sub and the pipeline) ranging from 20 to 26 seconds. Note that this lag, i.e. time to process network observations, does not directly impact the user perceived performance of the system, as this process is done asynchronously.

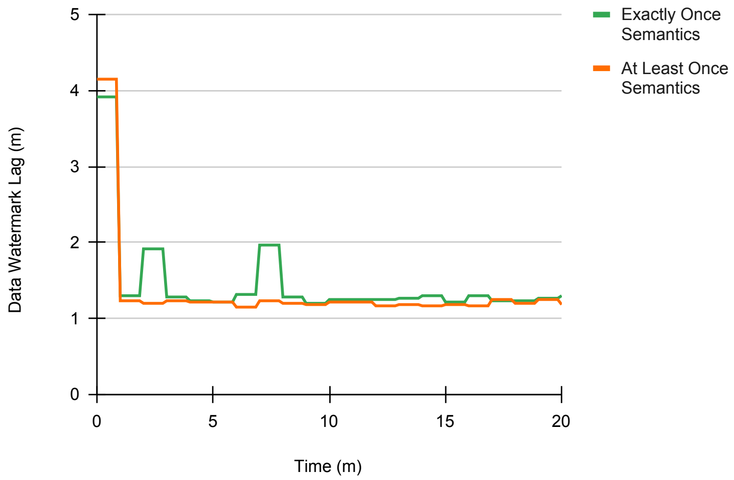

Figure 4.22: Data watermark lag per pipeline processing semantics.

4.6.6 Resource Utilization Analysis

As for resource utilization, we present plots for both CPU and memory in Figure 4.23 and Figure 4.24, respectively. We are able to observe that the pipeline ensuring *at-least-once semantics* utilizes more from both resources. More precisely, the pipeline with *exactly-once semantics* and the pipeline with at least once semantic achieve a mean CPU utilization of 16% and 21%, respectively, and a mean memory utilization of 4.6 GiB and 5.6 GiB, respectively. The observation of a higher processing rate upstream in the *at-least-once semantics* pipeline is a possible rationale for these resource utilization differences. Nonetheless, in spite of the utilization offset both pipelines present an identical trend on either resource.

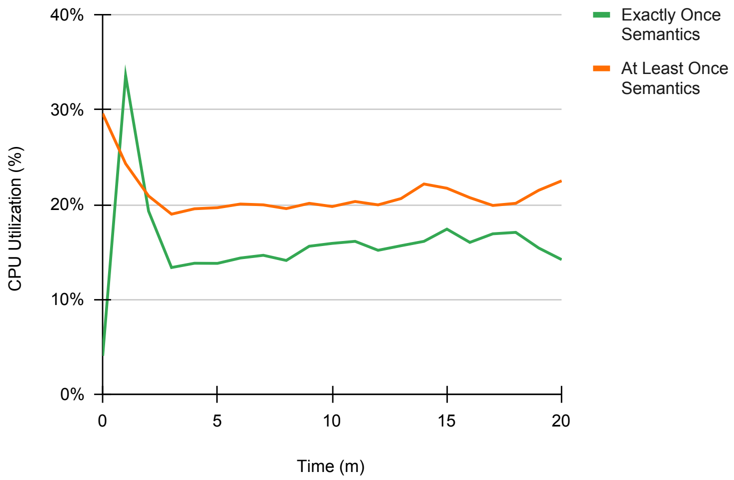

Figure 4.23: CPU utilization per pipeline processing semantics.

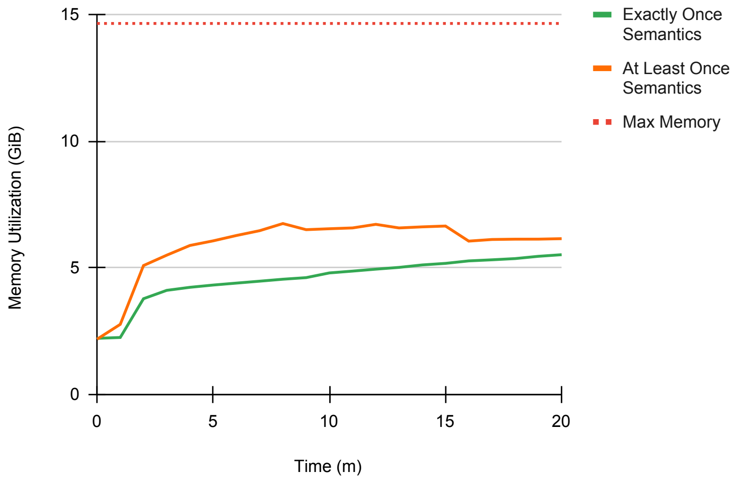

Figure 4.24: Memory utilization per pipeline processing semantics.

4.6.7 Summary

In sum, although the impact of using *exactly-once semantics* is significant on the achievable throughput, this impact is attenuated by the sum per key stage of the pipeline (combine function) which is naturally present in both pipelines, regardless of their processing semantics. Additionally, the main goal of our stream pipeline is to provide low-latency updates with the highest level of correctness, thus based on the median latency speedup of $1.05x$, we conclude that the ability to provide correct updates, by ensuring *exactly-once semantics*, for this specific workload outweights the minor performance impact.

4.7 Theoretical Model vs Practical Model Assessment

One of the major differences between the practical implementation of the CROSS City Cloud extension, detailed in Section 3.2, to support time-bound location proofs, and the theoretical model, described in Section 2.1.3, is the method of storing the Wi-Fi AP observations and the usage of time-based partitioned intermediate aggregate results. To properly quantify possible gains or losses when computing and retrieving the stable or volatile sets with our implementation and a *naive* theoretical implementation, we have developed a base solution of the model, as the baseline, and equivalent stable and volatile set queries. In the subsequent sections, we refer to the alternatives as the *"Practical Implementation"* and the *"Theoretical Implementation"*.

During the assessment, the measurements are performed over the time elapsed since the start of the query execution and the total *slot time* used by the query. The term slot is described in the "Query Plan Explanation" section[6] of the BigQuery documentation as *"an abstracted repre-*

[6] https://cloud.google.com/bigquery/query-plan-explanation#background

sentation of multiple facets of query execution, including compute, memory, and I/O resources", meaning the *"total slot time used"* metric provides an estimate of the individual query cost.

4.7.1 Experimental Setup

The theoretical implementation of the model was developed in BigQuery, involving the creation of a new dataset containing the tables which represent each relation (*Observations*, *Locations*, *Devices* and *Users*) as described in the model [CEP22], illustrated in Table 4.6. Additionally, similarly to the stream layer of our implementation, an Apache Beam pipeline was developed to extract, transform and load the signal observations of the test dataset, formerly described in Section 4.2, onto the aforementioned tables.

Table 4.6: Relations detailed in the theoretical model, adapted from [CEP22].

Relation	**Attributes**	**Description**
Observations	id, obsTime, location, device, signalType, transmitterId	Collected signal
Locations	id, name, coordinates	Points of Interest
Devices	id, name, userId	User mobile devices
Users	id, name	Users of the system

Regarding the time window parameters of the experiment for validating location proofs, we have set these to one week (2019-07-29 to 2019-08-04) and one day for the *epoch* and *period*, respectively, as referred in Section 3.1, with varying *spans* and points of interest.

4.7.2 Stable Set Computation

The stable set of a particular point of interest is computed over an *epoch* time window. As mentioned previously, the default value for the *epoch* parameter is of one week or seven days/*periods*. We have used the network observations correspondent to the week of 2019-07-29 to 2019-08-04 to compute the stable sets, since these are the sole consecutive seven days existent in the dataset. Six experiments across five repetitions were conducted for each one of the six distinct points of interest (Jerónimos, Comércio, Sé, Oceanário, Alvalade and Gulbenkian) among the test dataset. We have plotted the experimental results with regards to the *query elapsed time*, in milliseconds, and *slot time* used, in seconds, as presented in Table 4.7.

Table 4.7: Elapsed and Slot Time over Point-of-Interest per Implementation for the Stable Set Computation.

Implementation	Point-of-Interest	Elapsed Time (ms)	Elapsed Time Speedup (x)	Slot Time (sec)	Slot Time Speedup (x)
Theoretical	Alvalade	848.0	1.0	3.6	0.9
Practical		889.0		3.8	
Theoretical	Comercio	847.0	0.9	4.8	1.3
Practical		897.4		3.8	
Theoretical	Gulbenkian	798.2	1.0	4.0	1.2
Practical		797.4		3.4	
Theoretical	Jeronimos	791.4	0.9	4.0	1.1
Practical		871.4		3.8	
Theoretical	Oceanario	887.2	1.0	4.6	1.3
Practical		848.2		3.6	
Theoretical	Se	953.6	1.2	6	1.7
Practical		825.6		3.6	

For the *query elapsed time*, both implementations obtain similar results with the practical implementation being able to achieve on average a lower elapsed time for Gulbenkian, Oceanário and Sé while the theoretical implementation obtaining on average a lower elapsed time for the remaining three points of interest: Alvalade, Jerónimos and Comércio. The average difference between implementations is approximately 57 milliseconds (average speedup of approximately $1.1x$), with the minimum difference observed being only 0.8 milliseconds for the Gulbenkian point of interest while the maximum difference observed being 128 milliseconds for the Sé point of interest, equating to a speedup of approximately $1.2x$.

Regarding the *slot time* used, the practical implementation is able to achieve on average a lower *slot time* for five out of the six points of interest: Gulbenkian, Jerónimos, Comércio, Oceanário and Se. The theoretical implementation obtained on average a lower elapsed time for the remaining point of interest: Alvalade. The average difference between implementations is approximately 0.9 seconds (average speedup of approximately $1.2x$), with the minimum difference observed being 0.2 seconds for both the Alvalade and Jerónimos points of interest while the maximum difference observed being 2.4 seconds for the Sé point of interest, equating to a speedup of approximately $1.7x$.

Both implementations for the same amount of observations distributed across a single week achieve a similar amount of overall elapsed time, which is the metric with a greater influence on the user's perception of system performance. However, it is worth noting that in the practical implementation the stable set computation is done off the critical path onto a separate table, which necessarily depends on the existence of the stable set. But, this is done without impacting the volatile set computation, and consequently the user's experience. Thus, performance gains are expected to be more evident for the volatile set computations, presented in

Section 4.7.3. Moreover, as for the *slot time* experiment the results are on par with expectations, since both implementations are limited by two major functions (the *aggregate count* and *percentile*), nonetheless the practical implementation projects from the union of the tables of each *period*, instead of the double selection made by the theoretical implementation over the *obs* relation. Therefore, since the *obs* relation might contain observations from multiple different timestamps and points of interest this operation might present itself resource heavier or costlier.

4.7.3 Volatile Set Computation

The volatile set of a particular point of interest is computed over a *span* time window interval resultant from the client's time location claim and a *delta*. Since the volatile set computation depends on an existent stable set, we have used the network observations correspondent to the week of 2019-07-29 to 2019-08-04 to compute it, hence why the volatile set computations are performed over the day/*period* of 2019-08-19, as this is the closest day to the *epoch* used present in the dataset. Sixteen experiments across five repetitions were conducted for the Gulbenkian point of interest with varying *spans* and overall dataset size.

Varying Spans Analysis

The first eight experiments were performed with varying *spans* (1, 5, 10 and 15 minutes) derived from the respective *deltas* (30 seconds, 2.5, 5 and 7.5 minutes). As in Section 4.6 of the stream layer experiments, we have selected four realistically low deltas, in contrast to the maximum possible *span* which is a *period* (1 day). The Table 4.8 details the resulting experimental configurations including the claimed time interval used in the volatile set query. We have plotted the experimental results with regards to both the *query elapsed time* and *slot time* used, in seconds, presented in Table 4.9.

Table 4.8: Varying *Spans* and *Deltas* Volatile Set Experimental Configurations.

Point-of-Interest	***Span***	***Delta***	**Claimed Time Interval**
Gulbenkian	1 min	0.5 min	14:37:00 - 14:38:00
	5 min	2.5 min	14:35:00 - 14:40:00
	10 min	5 min	14:33:00 - 14:43:00
	15 min	7.5 min	14:30:00 - 14:45:00

Table 4.9: Elapsed and Slot Time per *Span* Interval for the Gulbenkian Volatile Set Query.

Implementation	**Span Interval (min)**	**Elapsed Time (sec)**	**Elapsed Time Speedup (x)**	**Slot Time (sec)**	**Slot Time Speedup (x)**
Theoretical	15	2.4	2.4	39.8	2.2
Practical		1.0		18	
Theoretical	10	2.0	2.0	43.8	2.5
Practical		1.0		17.8	
Theoretical	5	2.0	2.0	38.8	2.2
Practical		1.0		17.4	
Theoretical	1	2.0	2.0	34.4	2.0
Practical		1.0		17.2	

The practical implementation consistently outperformed the theoretical implementation for each span on both the *query elapsed time* and the *slot time* used metrics. Specifically regarding the *query elapsed time*, it was able to achieve on average a speedup of $2.1x$, with a minimum speedup of $2.0x$ and a maximum speedup of $2.4x$. Furthermore, with respect to the *slot time* used, it was able to achieve on average a speedup of $2.2x$, with a minimum speedup of $2.0x$ and a maximum speedup of $2.5x$.

These results match the expectations that the practical implementation would perform better than the theoretical implementation. It demonstrates the impact of computing the stable set off the critical path and its materialization onto a separate table, and the aggregation of network observations per period and point of interest on distinct tables. Note that the theoretical implementation requires on premise stable set computation, on top of maintaining all network observations individually on a single relation.

Varying Dataset Size Analysis

The last eight experiments were performed with varying input dataset sizes (8988, 12616, 22372 and 25404 KB) derived from the combination of the network observations of the remaining points of interest across the dates 2019-07-29 to 2019-08-19 existent in the dataset, as detailed in Table 4.10. The claimed time interval is maintained consistent throughout the experiments with the highest achievable accuracy, from 2019-08-19 14:37:00 to 2019-08-19 14:38:00 (1 min *span*). We have plotted the experimental results with regards to both the *query elapsed time* and *slot time* used, in seconds, presented in Table 4.11.

Table 4.10: Varying Input Dataset Size Volatile Set Experimental Configurations.

Input Points-of-Interest	Input Dataset Size (KB)
Gulbenkian	8988
Gulbenkian + Jerónimos + Comércio	12616
Gulbenkian + Jerónimos + Comércio + Oceanário + Alvalade	22372
Gulbenkian + Jerónimos + Comércio + Oceanário + Alvalade + Sé	25404

Table 4.11: Elapsed and Slot Time per Input Dataset Size for the Gulbenkian Volatile Set Query.

Implementation	Input Dataset Size (KB)	Elapsed Time (sec)	Elapsed Time Speedup (x)	Slot Time (sec)	Slot Time Speedup (x)
Theoretical	8988	2.0	2.0	32.6	1.9
Practical		1.0		17.6	
Theoretical	12616	2.0	2.0	33.2	1.9
Practical		1.0		17.8	
Theoretical	22372	2.0	2.0	39.4	2.2
Practical		1.0		17.8	
Theoretical	25404	2.0	2.0	44.2	2.5
Practical		1.0		17.8	

The practical implementation surpassed the theoretical implementation for each input dataset size on both the *query elapsed time* and the *slot time* used metrics. Specifically regarding the *query elapsed time*, it was able to achieve on average a speedup of $2.0x$, with a minimum and maximum speedups of $2.0x$. Furthermore, with respect to the *slot time* used, it was able to achieve on average a speedup of $2.1x$, with a minimum speedup of $1.9x$ and a maximum speedup of $2.5x$.

The obtained results match the expected volatile set computation performance gain. Additionally, particularly with these experiments we are able to observe a higher performance degradation with relation to the dataset size with the theoretical implementation, the additional input dataset size from 8988 KB to 25404 KB results in a $0.74x$ *slot time* speedup equating to a difference of 11.6 seconds, when compared to the $0.99x$ *slot time* speedup obtained with the practical implementation. This can be explained by the use of a single table to store all network observations as individual records, in the theoretical implementation. Although, this does not have a noteworthy impact on the elapsed time on our experiments, we should expect that for significantly larger sizes, above a certain threshold, the impact on the elapsed time would be noticeable. Additionally to further support the previous claim, based on these experiments we quantified the impact of the input size in the resulting BigQuery dataset size of each implementation plotted in Figure 4.25, which for the maximum increase in input size from 8988 to 25404 KB (+16416 KB) the practical implementation increases its dataset size in approximately

1.16 MiB, while the theoretical implementation requires an increase of approximately 10.9 MiB ($\approx 9.4x$).

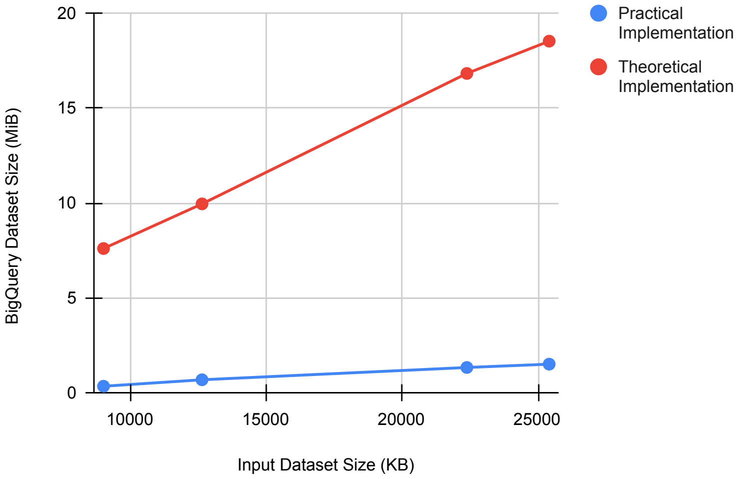

Figure 4.25: BigQuery Dataset Size per Input Dataset Size.

4.8 Summary

In this Chapter, we demonstrated the feasibility of providing location and time-bound proofs through the match of network observations scavenged by the prover devices and pre-computed stable and volatile sets, by achieving stable and volatile set match success rates of 61.11% and 63.89%, respectively. The performance and scalability analysis demonstrated that the system is able to scale horizontally, maintaining an acceptable performance level of under 100 ms under load with up to 50 concurrent users, at a 60% resource utilization. The trade-offs made in the performance and completeness of the pipeline solution with distinct processing semantics were analyzed and quantified, and we concluded that our solution is still able to provide real-time low-latency updates while enforcing a greater level of correctness. We evaluated the method used for integrating the Wi-Fi AP observations in the serving layer, with regards to client stable and volatile set query performance. Our solution significantly outperforms the baseline used by leveraging the stable set computation off the critical path. Most notably, it achieves an average speedup of ≈ 2.2x in perceived user location proof validation performance.

Chapter 5

Conclusion

We presented CROSS City Cloud, a cloud-native location certification system for consumer mobile applications, capable of producing and validating time-bound location proofs.

The ability to offer verifiable information with regards to a location time of visit strengthens the dependability of the location proofs generated. However, existing related work, more specifically the CROSS City v1 prototype, was only capable of validating the time of visit of a location claim through the augmentation of each supported location with purpose-built infrastructure. To address this limitation, we designed and implemented an architectural extension to the CROSS data management layer on top of a scavenger data analysis model, properly supporting location proofs with temporal granularity. The architectural extension can ingest, aggregate and integrate scavenged network observations in stable and volatile networks of a particular point of interest. Location claims made by tourists can be attested for the visited location and time of visit using solely the stable and volatile sets. A range of data storage and processing components, as well as prominent architectures were assessed, to select the most optimal solution for the requirements and objectives. The deployment of the system leverages modern virtualization technology and the existing services offered by the modern, commercial clouds, namely the Google Cloud public provider. By exploiting cloud computing and fine tuning each component, we were able to ensure the fulfilment of the production quality criteria including system scalability, reliability and observability of each component. We implemented a control plane for the automation of the configuration and deployment of the system, leveraging Infrastructure-as-Apps tools to conform to GitOps standards, such as being *Continously Reconciled*, i.e. keeping the service state always as declared by the operator. From the perspective of an operator, the control plane allows for an efficient management.

CROSS City Cloud achieved stable and volatile set match success rates of 61.11% and 63.89%, respectively, demonstrating its feasibility in providing location and time-bound proofs. Scalability and performance analysis demonstrated that the system is able to scale horizontally, maintaining the acceptable performance level of under 100 ms under load with up to 50 concurrent users, at a 60% resource utilization. Although *exactly-once semantics* significantly impact the achievable throughput, this impact is attenuated by the downstream operators and the pipeline parallelism. Thus, we assessed that our pipeline solution is still able to provide real-time low-latency updates while enforcing a greater level of correctness. We evaluated our method of integrating the Wi-Fi AP observations, through the usage of time-based partitioned intermediate aggregated views, with regards to client stable and volatile set query performance. This assessment was carried out against a baseline *naive* implementation of the theoretical scavenging model. Our solution significantly outperforms the baseline by leveraging the stable set computation off the critical path. Most notably, it achieves an average speedup of $\approx$ 2.2x in perceived user location proof validation performance.

In conclusion, we demonstrated the feasibility of embedding a time-bound location certification framework into public cloud computing technology to provide a Location-Certification-as-a-Service platform, for consumer mobile application use cases.

5.1 Achievements

We developed CROSS City Cloud, a cloud-native location certification system for consumer mobile applications, capable of producing and validating time-bound location proofs without additional infrastructure, leveraging public cloud services to deliver its services.

We implemented an architectural extension to CROSS City, based on the Kappa architecture, comprised of stream and serving layers. This extension is responsible for ingesting, transforming and integrating client scavenged Wi-Fi AP observations, to determine over particular time windows the stable and volatile networks of a given location. The stream layer handles the asynchronous communication between the CROSS REST API server and the pipeline that produces the intermediate views of aggregated network observations. The serving layer indexes the intermediate views and serves stable and volatile set query requests.

We designed and implemented a cloud deployment solution of the complete CROSS City architecture. The system was deployed to Google Cloud Platform leveraging the following services: Google Kubernetes Engine, Google Cloud Pub/Sub, Google Dataflow and Google BigQuery. The cloud solution components were tinkered to fulfill set production system criteria.

We implemented a CROSS City control plane, using ArgoCD and Crossplane, to automate the configuration and deployment of the system and its components, conforming to GitOps standards, aiding system operators manage and operate the CROSS City Cloud services.

We assessed the location and time-bound proof feasibility. We evaluated the performance and scalability of the domain layer. We evaluated the trade-offs made in performance and completeness of our stream processing pipeline solution with distinct processing semantics. We assessed the gains obtained with the use of the implemented time-based partitioned intermediate aggregated view method, with regards to stable and volatile set query performance.

5.2 Future Work

CROSS City Cloud supports a *smart tourism* use case, in which pedestrian type tourists can be more predominant. As a result, the window type and sizes utilized to aggregate network observations for the computation of the intermediate stable and volatile sets are fine-tuned to this sort of tourist and physical movements. Nonetheless, since low-latency updates are already supported through the utilization of a stream processing engine, window type and size parameter adjustments tailored to transportation tourism, such as buses and the iconic tuk-tuks, could be explored.

The monitoring layer of CROSS City Cloud provides system operators with a complete image of the system's health status. Nonetheless, further improvements can be made to this layer to enhance the degree of observability, for instance with the addition of logging and tracing capabilities. Logging would be useful to determine what each component was communicating at time of failure and further diagnose potential issues. Moreover, tracing can be leveraged to enable the ability to pinpoint problematic components of the system and constitute user traces from real-world executions to optimize system performance, since CROSS City is built on a microservice architecture,.

From an operational standpoint, CROSS City Cloud provides a control plane to manage and automate the configuration and deployment of its components. However, the CROSS City application business logic is expected to evolve over time, hence why the automation of its lifecycle can be enhanced. For example, the integration of a Continuous Integration and Continuous Delivery (CI/CD) pipeline automates the processes of testing the code, building an artifact and pushing it to a cloud registry.

System operators can utilize several user interfaces to interact with CROSS City Cloud, such as directly from the cloud provider, ArgoCD, Grafana, BigQuery, and the command-line interface (CLI). The combination of these interfaces into a single dashboard developed using proper front-end tooling, with the goal of offering both health status information and query capabilities, could improve the system operator user experience (UX).

5.3 Final Remarks

CROSS City Cloud is a time-bound location certification platform specifically built for the smart tourism use case and reliant on system operators to manage it. Nonetheless, we envision future scenarios where the platform presented in this work, can be utilized as a fully automatic location proof self service to be consumed anywhere by any user independently of its use case.

Bibliography

[ABH09] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented Database Systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[All90] Arnold O Allen. *Probability, statistics, and queueing theory.* Gulf Professional Publishing, 1990.

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[BCC^{+}17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless Computing: Current Trends and Open Problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[Ber14] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[Bre12] Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer*, 45(2):23–29, 2012.

[CCCDP13] Eyüp S Canlar, Mauro Conti, Bruno Crispo, and Roberto Di Pietro. Crepuscolo: A collusion resistant privacy preserving location verification system. In *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–9. IEEE, 2013.

[CDG^{+}08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[CEP22] Rui Claro, Samih Eisa, and Miguel L Pardal. Lisbon hotspots: Wi-fi access point dataset for time-bound location proofs. *arXiv preprint arXiv:2208.04741*, 2022.

[CIMS19] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The Rise of Serverless Computing. *Communications of the ACM*, 62(12):44–54, 2019.

[Cod89] Edgar F Codd. Relational Database: A Practical Foundation for Productivity. In *Readings in Artificial Intelligence and Databases*, pages 60–68. Elsevier, 1989.

[Cod02] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[CRM91] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186, 1991.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[FP18] Joao Ferreira and Miguel L Pardal. Witness-based Location Proofs for Mobile Devices. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018.

[FS19] Arnaldo Pereira Ferreira and Richard Sinnott. A Performance Evaluation of Containers running on Managed Kubernetes Services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 199–208. IEEE, 2019.

[Gra22] Ricardo António Augusto Grade. CROSS City Mobile Application: Gamified Peer-Based Location Certification Strategy. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Portugal, 2022.

[Gun06] Neil J. Gunther. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services.* Springer-Verlag, Berlin, Heidelberg, 2006.

[HHLD11] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL Database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference*, volume 8, 2010.

[ISO11] ISO. *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).* International Organization for Standardization, Geneva, Switzerland, 2011.

[JPA$^+$12] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A Survey and Comparison of Relational and Non-Relational Database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[JSL$^+$11] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached Design on High Performance RDMA Capable Interconnects. In *2011 International Conference on Parallel Processing*, pages 743–752. IEEE, 2011.

[Kle17] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* "O'Reilly Media, Inc.", 2017.

[Kre14] Jay Kreps. Questioning the Lambda Architecture. `https://www.oreilly.com/radar/questioning-the-lambda-architecture/`, 2014. Accessed: 01-12-2021.

[LB10] Jeong Heon Lee and R Michael Buehrer. Location spoofing attack detection in wireless networks. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–6. IEEE, 2010.

[Lin17] Jimmy Lin. The Lambda and the Kappa. *IEEE Internet Computing*, 21(05):60–66, 2017.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[Mai19] Gabriel Antunes Maia. CROSS: loCation pROof techniqueS for consumer mobile applicationS. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Portugal, 2019.

[Mar11] Nathan Marz. How To Beat The CAP Theorem. `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`, 2011. Accessed: 01-12-2021.

[MCP20] Gabriel A Maia, Rui L Claro, and Miguel L Pardal. CROSS City: Wi-Fi Location Proofs for Smart Tourism. In *International Conference on Ad-Hoc Networks and Wireless*, pages 241–253. Springer, 2020.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2011.

[Mil68] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, 1968.

[MMPR11] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. Totp: Time-based one-time password algorithm. Technical report, 2011.

[Möd20] Stefan Möding. Analyze system scalability in R with the Universal Scalability Law. 2020.

[NPP+17] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[Pah15] Claus Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[See09] Marc Seeger. Key-Value stores: a practical overview. *Computer Science and Media, Stuttgart*, 9:1–21, 2009.

[SKS+02] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 5. McGraw-Hill New York, 2002.

[SR86] Michael Stonebraker and Lawrence A Rowe. The Design of Postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[SRH90] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE transactions on knowledge and data engineering*, 2(1):125–142, 1990.

[WM15] James Warren and Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[WPZM16] Xinlei Wang, Amit Pande, Jindan Zhu, and Prasant Mohapatra. Stamp: Enabling privacy-preserving location proofs for mobile users. *IEEE/ACM transactions on networking*, 24(6):3276–3289, 2016.

[ZC11] Zhichao Zhu and Guohong Cao. Applaus: A privacy-preserving location proof updating system for location-based services. In *2011 Proceedings IEEE INFOCOM*, pages 1889–1897. IEEE, 2011.

Appendix A

Data Storage Models In Depth

In this section we will discuss in more detail the specific implementations of each data storage model, initially introduced in Section 2.2, general use cases of the implementation and mention examples of other implementations. Each database type analysis in this section will be structured with a detailed analysis of a specific implementation, general use cases of the analysed implementation, and examples of other implementations.

A.1 Relational

Regarding PostgreSQL's main characteristics[1]:

Transactions *PostgreSQL* does support ACID semantics for transactions with atomicity and consistency being guaranteed by both a rollback mechanism and a MVCC (Multiversion Concurrency Control) locking method, multiple isolation levels provided by a snapshot "filter" and durability being ensured by the storage manager.

Query Language PostgreSQL utilizes the standard Structured Query Language (*SQL*). As previously stated, PostgreSQL stores data in tables, which are collections of rows, each of which contains a set of columns (attributes). Each row in a certain table has the same set of columns, as defined by the table's schema. *Data Definition statements* (DDS) are used to create and modify those tables. DDS also provides additional control over how data is stored in tables through table partitioning and triggers. DDS also provides further control over how the data will be stored in the tables with table partitioning and triggers. *Data Manipulation statements* (DMS) are used to insert, update and delete data from tables. To retrieve data from a table, PostgreSQL supports projecting specific columns from a set of rows (SELECT clause), retrieving rows matching a given multi-conditional predicate (WHERE clause), grouping rows

[1] https://www.postgresql.org/docs/14/index.html

for aggregate functions (GROUP BY clause), sorting rows (ORDER BY clause) and limiting the amount of rows returned (LIMIT clause). Tables can be joined according to a variety of rules including inner, outer and cross-joins. Data from joined tables may also be queried. Updated results of a query can also be kept in a materialized view. PostgreSQL provides the ability to create multiple types of indexes on tables (B-Tree, Hash, GiST, SP-GiST, GIN, BRIN), as a means to improve the performance of certain queries with the introduction of overhead.

Replication *PostgreSQL* supports unidirectional *Primary-Standby* replication for high availability, and some level of fault-tolerance. More specifically, a cluster may only contain a single primary node, which is responsible for receiving data modifications and communicating such data updates to the backup servers, either asynchronously or synchronously. Standby servers can answer read requests and may only be allowed to do local data updates, which will not be further replicated. In addition, standby servers can accept replication connections and stream WAL (Write-Ahead Logging) records to other standbys, which is referred to as *"Cascading Replication"*. Regarding failovers, it is worth to note that *PostgreSQL* does not provide the system software needed to detect a failure on the primary node and alert the standby nodes.

Partitioning *PostgreSQL* supports three types of basic table partitioning, in order to split a large table into smaller physical pieces, as a means to improve query performance. Range Partitioning which is a key column or combination of columns is used to split the table into "ranges.". List Partitioning where the table is partitioned by stating which key values occur in each partition explicitly. Hash Partitioning where each partition of the table is achieved by specifying a modulus and a remainder. However, it is important to note that *PostgreSQL* does not provide native support to sharding, a horizontal scalability solution where each partition is held on a separate database server instance.

Storage Engine *PostgreSQL* utilizes Write-Ahead Logging (WAL), in order to first record any changes made to data files, containing tables and indexes, in a sequential manner before flushing these to permanent storage. It also utilizes checkpoints, points from which it is assured that the heap and index data files have been updated with all information recorded prior to that checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a specific checkpoint record is recorded to the log file.

General use cases of PostgreSQL:

- **OLTP database**: Online Transaction Processing (OLTP) applications are concerned with transaction-oriented processes that must be executed at a high rate. Relational databases are highly suited for OLTP applications as they allow for inserting, updating, and deleting data across several tables, while also allowing frequent queries. Since PostgreSQL is entirely ACID compliant, it is suited for OLTP workloads;

- **Geospatial database**: PostgreSQL supports geographic objects when used with the PostGIS extension and may be used as a geospatial data store for location-based services and geographic information systems (GIS). PostGIS is a highly standard-compliant and de-facto standard in the Open Source GIS industry, providing hundreds of functions for processing geometric data in multiple formats;
- **Federated hub database**: The Foreign Data Wrapper extension for PostgreSQL allows it to connect to other data sources, including NoSQL databases, and operate as a federated hub for polyglot database systems.

Mainstream examples of relational databases:

- **MySQL**: Open-source RDBMS, also available under multiple proprietary licenses, written in C and C++;
- **Microsoft SQL Server**: Microsoft's relational database management system, under a proprietary license, written in C and C++;
- **IBM Db2**: Relational database management system designed for performance in transactional workloads.

A.2 Key-value

Regarding Redis' main characteristics[2]:

Transactions *Redis* can not handle ACID transactions of the kind found in relational databases, instead, it can only support lightweight transactions with no rollback mechanism and an optimistic locking check-and-set (CAS). All commands in a transaction's execution block are guaranteed to be isolated, atomic, and durable.

Query Language Redis lacks support for a more traditional and robust query language due to its simplified key-value data model. With this in mind, it lacks concepts such as *Data Definition* and *Data Manipulation* statements. Redis does provide commands for setting and getting a value of a specified key. Since Redis supports a variety of data structures as values, specific operations related to each data structure are also supplied. *Redis Lists*, for example, includes methods for inserting elements into the list at the head or tail. In addition, Redis supports cursor-based queries with a few aggregation functions, such as "COUNT", which limits

[2] https://redis.io/documentation

the amount of elements returned. It is worth noting that Redis does not handle more complex queries such as joins.

Replication Replication in *Redis* can be used as a means to achieve high availability and durability. *Redis* replication is built on a simple primary-backup replication scheme, in which *Redis* replicas are perfect copies of a single master instance. When the link between the master and the replica breaks, the replica can immediately reconnect to the master and seek to be an identical copy of it, regardless of what happens to the master. Client's write requests must always go to the primary node, whereas read requests can be routed either to the primary or any of the replicas. It's worth noting that *Redis*' basic replication system detailed here does not enable any type of automated failover in the event of a master's failure, just a manual failover. *Redis* provides by default asynchronous replication, off the critical path, to maintain request's low latency and high performance. However it also able to provide synchronous replication to ensure that a certain number of acknowledged copies are obtained from *Redis* instances, but it does not transform a set of *Redis* instances into a CP system with strong consistency, because acknowledged writes can still be lost during a failover.

In order for a *Redis* primary-backup deployment to achieve high availability without the need for human intervention, *Redis* requires the deployment to run *Redis Sentinel*, which is a distributed monitoring system. *Redis Sentinel*, on top of its monitoring layer is able to provide an automatic failover solution.

Partitioning *Redis Cluster* is *Redis*' solution for sharding data from a dataset among several Redis nodes automatically. *Redis Cluster* does not utilize consistent hashing, but rather a type of sharding in which each a key is part of what they term a hash slot. *Redis Cluster* has a total of 16384 hash slots, and to get the hash slot of a given key, they obtain the *CRC16* of the key modulo 16384. Every node in a *Redis Cluster* is in charge of a fraction of the hash slots, and can be replicated using the same primary-backup type replication scheme described above. This enables some level of availability during partitions, when a subset of nodes fails or is unable to communicate. However, in the event of a failure in a majority of the master nodes, the cluster ceases to function.

Storage Engine Despite the fact that *Redis* operates with its data structures directly in-memory, in other words with its dataset in-memory, it also supports two separate persistence models:

- **RDB** (Redis Database): At preset intervals, it takes point-in-time snapshots of the dataset and saves them to disk;

- **AOF** (Append Only File): The AOF persistence logs every write operation received by the server in an append-only mode, which is then replayed upon server restart to reassemble the original dataset. When the log reaches a particular size, Redis can rebuild it in the background to the smallest sequence of commands required to recreate the current dataset in memory.

RDB is a compact single-file point-in-time representation of the dataset that is ideal for backups. However it is not able to provide the same level of durability as AOF. On top of both of these models, *Redis* also supports a combination of both RDB and AOF or no persistence at all.

General use cases of Redis:

- **Caching**: A cache layer has the set objective of decreasing an application's latency and increase throughput. This can be achieved by keeping frequently accessed data on ephemeral, but very fast storage. Redis as an in-memory key-value store is a popular solution for a distributed caching solution. Redis provides the ability to perform sub-millisecond response times at scale through sharding, while maintaining high availability through replication;

- **Session Management**: Session data typically refers to user data from a certain session state, for example in the form of user profiles, login credentials and other user-specific personalization information. Reading and writing session data with each user interaction must be done in a way that does not compromise the user experience. When a user disconnects and thus terminates a session, session data must be preserved for future use. Session state may be maintained in Redis as a key-value pair, with the user identifier as the key and the session data as the value. Applications can only read and write to the Redis in-memory session store, and if durability is required, they can use Redis' persistence options;

- **Messaging**: Modern applications built on a microservice architecture are composed of several loosely coupled services that must communicate with one another. Redis offers data structures such as lists, sorted sets, and hashes, which are used to implement message queues. Redis also provides a publish subscribe messaging protocol that may be used to broadcast live notifications within a system.

Mainstream examples of key-value databases:

- **Memcached** [JSL+11]: Key-value distributed memory object caching system. Most used for caching results of database calls, API calls or any other data;

- **Tokyo Cabinet**: A key-value database that supports 3 modes of operation: hashtable mode, b-tree mode, and table mode, written in C;
- **etcd**[3]: Strongly consistent distributed key-value store, implemented in Go.

A.3 Column-oriented

Regarding Cassandra's main characteristics[4]:

Transactions *Cassandra* does not employ relational database type ACID transactions with rollback or locking methods, instead it opts for atomic, isolated, and durable transactions with eventual/tunable consistency, allowing the user to choose how strong or eventual the consistency of each transaction is. At the row level, *Cassandra* supports atomicity and isolation, however it sacrifices transactional isolation and atomicity with high availability and rapid write speed. *Cassandra* also does not allow joins or foreign keys since it is a non-relational database.

Query Language *Cassandra Query Language (CQL)* offers a similar model and syntax to SQL, which was discussed in Section 2.2.2. CQL stores data in tables, the schema of which dictates the layout of the data in the table, and tables are stored in keyspaces. *Data Definition statements* are used to create, modify and remove keyspaces and tables. *Data Manipulation statements* are used to insert, update, delete and query data from tables. When querying data, CQL allows retrieving rows matching a specific multi-conditional predicate (WHERE clause), grouping results for aggregate functions (GROUP BY clause), ordering results (ORDER BY clause) and limiting the amount of results returned (LIMIT BY clause). It is important to note that data from different column families cannot be joined in CQL. CQL provides the ability to create secondary indexes on tables, allowing queries to leverage such indexes. In order to maintain a set of rows from a table specified in a SELECT statement, materialized views can be created.

Replication Replication in *Cassandra* is used as a means to achieve high availability and durability. Each data item in a partition, with a key belonging to a certain token range, is replicated across N distinct nodes in the cluster, where N is the replication factor pre-configured. *Cassandra* provides several distinct replication strategies, in order to determine which replicas are selected for a certain token range, including *"Rack Aware"*, *"Rack Unaware"* and *"Datacenter Aware"*. Note that all replicas are equally important, there is no primary or master replica. *Cassandra* contains two types of nodes in a cluster:

- **Coordinators**: The coordinator is in charge of replicating the data items within its token

[3] https://etcd.io/

[4] https://cassandra.apache.org/doc/4.0/

range. The coordinator not only stores each key within its range locally, but also replicates them at N-1 nodes in the ring;

- **Non-Coordinators**: Nodes which act as replicas for data items within a particular range of keys;
- **Leader**: Apart from the two above, a single node in the cluster will also be elected as the leader, through a system called *ZooKeeper* [HKJR10]. When nodes join the cluster, they contact this node, who informs them of the token ranges for which they are replicas.

Any node in the cluster can receive client read or write requests. Receiving nodes operate as a proxy between the client application and the nodes that own the data being requested, acting as the coordinator for that particular client operation. Based on how the cluster is setup, the coordinator selects which nodes in the ring should receive the request and waits for acknowledgements of a specific number of replicas, based on the consistency guarantees requested by the client.

Cluster membership information and other system-related control state is regularly sent between nodes using Gossip, a peer-to-peer communication protocol. When feasible, Cassandra utilizes this information to avoid routing client requests to inaccessible nodes.

Partitioning *Cassandra* provides horizontal scalability by utilizing a hash algorithm to partition all data stored in the system. Cassandra's hashing technique, in particular, is *consistent hashing* with order preservation. The output range of the hash function is viewed as a *ring*, meaning the largest hash value wraps around to the smallest hash value. Within this region, each node in the system is assigned a random value that indicates its token/location on the ring. Each data item is allocated to a coordinator node, which is determined by hashing the data item's key to get its token/location on the ring, then traveling the ring clockwise to identify the first node with a position greater than the item's position.

Storage Engine *Cassandra* makes use of Google's *Bigtable* data and storage engine model. Cassandra's storage engine provides commit logs, which are append-only logs recording all Cassandra node modifications. Every write operation, also referred to as a mutation, to Cassandra is first written to a commit log in a sequential manner. Only after being successful does the data get written and indexed in-memory to a *Memtable* structure. This ensures both durability and recoverability, in the event of an unexpected shutdown. Only once a certain size threshold is met do the in-memory structures get flushed into disk, becoming immutable *SSTables*, which are data files used by *Cassandra* in order to persist data on disk. Cassandra also initiates compactions, which merge several *SSTables* into a single table.

General use cases of Cassandra:

- **Time-series data**: Time-series data are measurements performed at consistent time intervals. This type of data tends to come at a high frequency and has the potential to generate vast volumes of data, a typical use case is Internet of Things. Cassandra is highly efficient with writes providing a higher throughput. Since time-series data tends to involve write heavy workloads Cassandra is a good fit for it. As large amounts of data get integrated Cassandra is able to scale and maintain performance, due to its sharding methods;

- **E-commerce**: E-commerce companies rely on the reliability and availability of their services to generate a profit, which is especially true during peak hours. They must also be able to scale their online inventory cost-effectively. Inventory data may also easily reach massive amounts of data. And, in order to adapt to an ever-changing market, they must be able to continually adjust their product offerings. Cassandra supports multi-master replication for high availability and its peer-to-peer architecture allows data to reside in multiple regions closer to any particular customer. Cassandra also provides cost-effective sharding methods for horizontal scalability;

- **Recommendation Engine**: Personalization and recommendation engines are widely used in modern applications and websites to tailor the experience to each unique user. Cassandra is a suitable match for this sort of application since it demands high write throughput with low latency and real-time analytics across the dataset. Similarly to the previous use cases, as the dataset grows Cassandra is able to horizontally scale through sharding.

Mainstream examples of column-oriented databases:

- **BigTable** [CDG$^+$08]: Distributed storage system for managing structured data designed for large scale, most notably web indexing, implemented in C++;

- **Hypertable**: DBMS inspired by Google Bigtable, meant to run on top of a distributed file system such as the Apache HDFS, GlusterFS or the CloudStore Kosmos, implemented in C++;

- **Apache HBase**: Column-oriented database built on Apache Hadoop and BigTable concepts, written in Java.

A.4 Document-oriented

Regarding MongoDB's main characteristics[5]:

Transactions To begin, it's crucial to note that *MongoDB* adheres to ACID principles, which is critical for understanding the ideas that follow. Each transaction can adopt different transaction-levels regarding the read preference, read concern and write concern. Read preference defines to which node a client's read requests is routed to first, either to the primary or any secondary. Read and write concerns allow the client to have a strict control of the consistency and isolation required for the application, defining the number of distinct node acknowledgments required for a certain, read or write, request to accumulate before returning back to the client.

Query Language *MongoDB* has a distinct language model from SQL, explored in Section 2.2.2, with a JSON-like syntax. MongoDB stores data in collections, which are groups of documents. Collections are not required to have a defined schema, meaning that not all documents in a collection need to have the same fields. It is worth mentioning, though, that schema validation may be used to lock down the schema of a given collection. With this in mind, MongoDB lacks the traditional concept of *Data Definition statements*. MongoDB, on the other hand, offers four types of operations to create, read, update and delete documents. When querying data, MongoDB supports retrieving documents matching a given multi-conditional predicate and project specific fields. In addition, MongoDB has a plethora of aggregation functions including grouping, sorting and limiting document results. In contrast to CQL described in Section 2.2.4, MongoDB is able to performs left outer joins between collections. MongoDB provides the ability to create secondary indexes (single-field, compound and multi-key) on collections.

Replication Replication in *MongoDB* is accomplished using replica sets. A replica set is a group of *mongod* processes that keep track of the same data set. Replica sets are able to provide redundancy and high availability, on top of a level of fault tolerance against the loss of a single database server. A replica set consists of many data-bearing nodes and, if desired, one arbiter node. One and only one member of the data-bearing nodes is designated as the primary node, while the others are designated as secondary nodes. Each node type has a different role in the replication process:

- **Primary**: By default, receives all write and read operations. Records all changes to its data set in its operation log (*oplog*);

- **Secondary**: Replicates the primary's *oplog* and applies the operations to its data set, such that it matches with the primary's data set. In the event that the primary is not

[5] https://docs.mongodb.com/manual/

available, an eligible secondary will hold an election to elect a new primary;

- **Arbiter**: This node does not have a copy of the data set and cannot become a primary. However, it is able to cast a vote. When a cost limitation prevents the installation of another data-bearing node, arbitrator nodes prove useful. To be noted however that, if a secondary is unavailable or behind in a three-member primary-secondary-arbiter (PSA) architecture, requests with write concern "majority" might create performance concerns.

Secondary members of a replica set synchronize data from some other member, referred to as their *sync sources*, to replicate the primary's *oplog*, and therefore keep up-to-date copies of the shared data set. This synchronization method requires secondaries to select their synchronization sources, which happens right after their entrance on the replica set. After that, *sync from* sources sends a continuous stream of *oplog* entries to their syncing secondaries.

Partitioning The primary objective of *MongoDB*'s system is to enable horizontal scalability via sharding. This approach is useful for dealing with huge data volumes or high throughput operations that cannot be handled by a single server. In essence, the cluster data is separated into smaller pieces for each replica set, which not only helps the system handle more operations, but it also provides an increase in storage capacity. As previously mentioned, *MongoDB* has a sharded cluster query router called *mongos*, which serves as a bridge between the applications and the sharded cluster, determining the correct shard to which any request should be forwarded to. To retrieve the best performance, efficiency and scalability of a sharded cluster, shard keys were introduced, which consist of a field or multiple fields in the documents. Shard keys are used to make the distribution of datasets across shards, which are then used by both methods of sharding supported by *MongoDB*: *Hashed Sharding* and *Ranged Sharding.*

Storage Engine *MongoDB*'s storage engine is the main component in charge of data management, more specifically it controls how data is saved, both in memory and on disk. *MongoDB* offers two storage engines to fulfill different workloads. WiredTiger Storage Engine which is well-suited for most workloads, and having as its main characteristics the document-level concurrency model, checkpointing and compression. In-Memory Storage Engine which keeps documents in-memory, rather than on disk to ensure a more predictable data latency. To be noted that this particular storage engine does not persist data after process shutdown.

To further increase the durability guarantees, *MongoDB* utilizes *write ahead logging* to on-disk journal files, providing resilience in case of a failure. The *WiredTiger* storage engine already uses checkpoints to maintain a consistent view of data on disk and to allow *MongoDB* to recover from the last checkpoint. However, if *MongoDB* were to crash suddenly in between checkpoints, it would not be able to recover data lost after the last checkpoint, that is why journaling is necessary. There is no separate journal when using the in-memory Storage Engine, since its data is kept in memory.

General use cases of MongoDB:

- **Content Management Systems**: Content management systems (CMS) store information assets and their related metadata. This type of content needs to be served to various applications, including websites, online publications and archives. With this in mind, CMS platforms are required to accommodate any kind of data and as large amounts of content are integrated the platform should also scale, either horizontally or vertically. The JSON-like document model and rich query language of MongoDB enables applications to store and query several content types with varying attributes. To scale horizontally, MongoDB supports sharding;

- **Mobile Applications**: Mobile apps must meet different, more strict criteria than conventional computer applications. Such criteria include a more efficient battery consumption and the ability to function offline. MongoDB provides a cross-platform mobile database named *Realm*. MongoDB Realm is able to function offline locally as it persists data on-disk, and can sync data whenever multiple devices are connected. Realm also supports automatic conflict resolution by merging data changes via timestamps and operational transforms;

- **Internet of Things (IoT)**: Internet of Things refers to the concept of embedding sensors into physical objects which connect to the Internet as a means to exchange their collected data. IoT projects tend to be composed of multiple distinct objects acting as sources of data. With this in mind, IoT platforms are required to ingest and process large volumes of data over time. As vast amounts of data are integrated, the platform should remain highly available and scale both horizontally or vertically. MongoDB natively supports time series data, provides primary-backup replication to ensure highly available servers and allows sharding to scale horizontally.

Mainstream examples of document-oriented databases:

- **Couchbase**: JSON document-oriented database built for versatility, performance, scalability and financial value across cloud deployments, implemented in C;

- **Firebase Realtime Database**: Google's cloud-hosted realtime document store, popular for mobile and Internet Of Things applications;

- **Google Cloud Firestore**: An auto-scaling document-oriented database built for mobile and web apps that allows for cross device synchronization.

Appendix B

Service Operation

In this Chapter, we go over the operational aspects of CROSS City Cloud from the standpoint of the *system operators*. We detail the specific methodologies implemented to continuously integrate and deploy each element of CROSS, in an automated manner. Moreover, we integrate a fine-grained access control for each cloud service utilized.

B.1 Automated Configuration and Deployment

An important aspect of operating cloud services efficiently is the automation of the configuration and deployment of the system and its components, as these will inevitably evolve over time. CROSS City Cloud is comprised of two main elements, at a high level: the infrastructure resources and the business logic.

To automate the creation of the infrastructure (resource provisioning), we can either leverage Infrastructure-as-Code (IaC) or Infrastructure-as-Apps (IaA). Several tools are available for IaC such as Terraform[1] and Pulumi[2]. For IaA, commonly used tools include ArgoCD[3] and Crossplane[4]. We based our decision between IaC and IaA on the fulfillment of the GitOps principles. The GitOps principles are defined in OpenGitOps[5], which is a standardized approach to implementing GitOps through a set of open-source standards and best practices:

- *Declarative* - A system managed by GitOps must have its desired state expressed declaratively;
- *Versioned and Immutable* - Desired state is stored in a way that enforces immutability, versioning and retains a complete version history;

[1] https://www.terraform.io/

[2] https://www.pulumi.com/

[3] https://argoproj.github.io/cd/

[4] https://crossplane.io/

[5] https://opengitops.dev/

- *Pulled Automatically* - Software agents automatically pull the desired state declarations from the source;

- *Continuously Reconciled* - Software agents continuously observe actual system state and attempt to apply the desired state.

A declarative IaC configuration tool such as Terraform does not comply with the *Continuously Reconciled* principle, since these declarative formats are typically applied once or on explicit change to the desired state via some kind of Continuous Integration/Continuous Delivery (CI/CD) pipeline. And, these pipelines are not capable of monitoring the state for drift, i.e. for differences between the current and the desired state. For example, if a certain system operator alters the state of the cloud resources directly through the cloud provider this drift of state will not be corrected and not even detected.

To overcome the raised *Continuously Reconciled* issue, we have opted to leverage IaA by combining two tools: ArgoCD and Crossplane. ArgoCD is a GitOps continuous delivery tool. ArgoCD utilizes a Kubernetes controller to continually monitor the state of all resources under its management and compare it to the desired states specified in a Git repository. Crossplane is an open-source project that allows the provisioning and management of any kind of cloud resource via the Kubernetes API. ArgoCD has the concept of an application that includes a desired state configuration comprised of the intended destination and policies for syncing and managing its resources. When an application is deployed, ArgoCD continually monitors the actual state of the application and compares it to the desired state through its Kubernetes controller, as previously mentioned. When there is divergence, whether due to drift or a new desired state, there will be a reconciliation. In sum, by combining both the ArgoCD application concept and the Crossplane Custom Resource Definitions (CRDs), we are able to define infrastructure as applications and satisfy all the GitOps principles.

From an implementation standpoint, we maintain a separate GKE cluster, as the CROSS City Cloud Control Plane containing the Crossplane CRDs, the GCP Provider and an ArgoCD controller. Moreover, a Github repository is preserved comprised of both the Crossplane cloud service definitions and the applications as standard Kubernetes manifests. The ArgoCD controller establishes a connection between this repository and the main GKE cluster continually syncing the state as defined in the repository. Note that, in the event that the connection with the repository is down, then continuous syncing will also be down. To mitigate this, the repository can be replicated across multiple version control hosting solutions, such as GitLab[6] and

[6] https://about.gitlab.com/

Bitbucket[7], Github is being used as an example implementation. Another advantage of adopting Crossplane is that we can remain cloud-provider agnostic while maintaining homogeneity by leveraging the Kubernetes API as our single interface for orchestrating both infrastructure and applications. Figure B.1 illustrates the CROSS City Cloud control plane composed of ArgoCD and Crossplane.

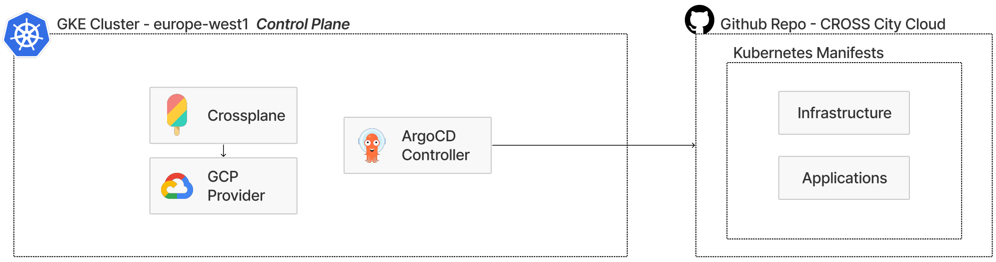

Figure B.1: The CROSS City Cloud Control Plane.

The automation of the build and deployment of the application business logic is not in the scope of this work, however the integration of a CI/CD pipeline, leveraging for example GitHub Actions, could be explored to achieve this goal. The pipeline would be required to continuously test the code, build the artifact and push it to the project's cloud registry.

B.2 Secure Cloud Environment

Although the SureThing project already includes a variety of use case projects to demonstrate the usefulness of location certification. CROSS City Cloud is the pioneer in deploying its services on the cloud. Therefore, we must establish common foundations of the SureThing project on Google Cloud by creating a manageable secure cloud environment, that can be reused by other projects. Specifically, we should enforce a strict access control to the several cloud services used, as formerly mentioned in Section 3.3.2, at the project cloud level. In Google Cloud Platform[8], fine-grained access control to cloud resources is done centrally via IAM. IAM allows the adoption of the security principle of least privilege, which states that every service or user should only have the permissions required for its legitimate purpose. IAM is able to handle authentication and authorization based on three main concepts:

- Principal: A principal is the Google cloud identity of a specific entity, typically associated with an email address as the identity;

[7] https://bitbucket.org/

[8] Due to the similarities across cloud providers, we explore the GCP security services in more detailed as a specific example, however these also apply to other providers.

- Role: A role is a collection of permissions which determine the set of operations allowed on a specific resource;
- Policy: A policy is a collection of bindings of one or more principals to specific roles.

The process of granting access to a specific resource involves the three concepts described above. First the principal attempting to access a certain resource is authenticated through the verification of its associated identity (authentication). The IAM policy of the resource, containing the principal's role, is then assessed to determine whether the action is allowed (authorization).

Google Cloud resources are organized hierarchically as organization, folders, projects and resources. The organization which is the root node in the hierarchy. Folders which are children of the organization. Projects which are either children of the organization or of a folder. Resources for each cloud service which are descendants of projects. IAM policies can be defined at any level of the resource hierarchy and each resource inherits the policies of all of its parent resources.

In our particular use case, the resource hierarchy, illustrated in Figure B.2, is composed of the *SureThing* organization, *CROSS City Cloud* folder and separate projects for distinct development environments. This hierarchy can be extended to integrate additional applications as different folders. New applications would automatically inherit IAM policies set at the organization level, but not at the folder level which is crucial to preserving policy consistency and scalability.

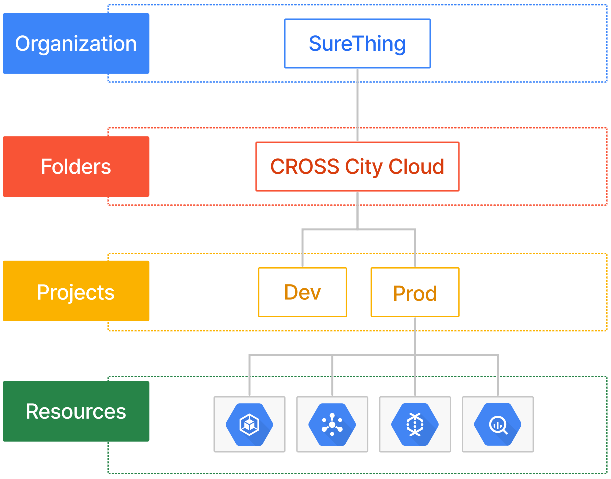

Figure B.2: Overview of the SureThing resource hierarchy on Google Cloud Platform.