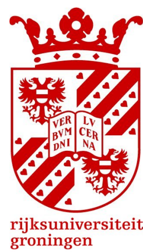
COST AND PERFORMANCE TRADE-OFF EVALUATION IN MICROSERVICES IMPACTED BY THE CAP THEOREM LIMITATIONS

DEEPSHI GARG



MASTER'S THESIS

Department of Computing Science
Faculty of Science and Engineering
University of Groningen

SUPERVISORS:
Dr. V. Andrikopoulos
Diarmuid Kelly
Prof. dr. A. Lazovik

September 2021 - January 2022

Deepshi Garg: *Cost And Performance Trade-off Evaluation In Microservices Impacted By The CAP Theorem Limitations,* Master's Thesis © September 2021 - January 2022

SUPERVISORS:
Dr. V. Andrikopoulos
Diarmuid Kelly
Prof. dr. A. Lazovik

LOCATION:
Groningen

TIME FRAME:
September 2021 - January 2022

ABSTRACT

A microservices architecture is effectively a distributed system, with each microservice being a single (or a collection of) nodes(s). Thus, the principles and limitations of the CAP theorem also apply to microservices, and especially when these are used to implement data-focused applications such as federated learning. This project provides a research framework to refine the design of data-focused microservices architecturally and infrastructurally while keeping in mind their CAP theorem constraints. We present an elaborate procedure to analyse different design alternatives for a microservice application with CAP theorem impositions, evaluate the trade-offs amongst different feasible solutions in terms of performance and operational expenses, and arrive at an optimal architecture in terms of implementation and infrastructure cost viability. Additionally, we exemplify the whole process with three examples from a case study in an industrial context, each lying on a different edge of the CAP triangle. Important thing to note here is that we only use CAP theorem to set the context for the research, by classifying the microservices as CA, CP or AP.

ACKNOWLEDGEMENTS

First and foremost, I thank my supervisor, Dr. Vasilios Andrikopoulos. He has been a very supportive supervisor all throughout. He has been especially helpful in defining sequential steps and a clear timeline for the research. He gave me biweekly goals, and reviewed the progress accordingly. He always inspired me to dig deeper into the problem, and find multi-dimensional solutions to it. For instance, he has been very keen on exploring architectural solutions as well, especially when I was limiting my focus onto infrastructural solutions. He consistently encouraged me to find supporting literature for every statement I make, and validate if it is in the correct direction or not. This saved me a lot of time by exploring only the relevant solutions, and not going into any random solution which crosses my mind. Consequently, this not only ensured that my project is well researched into the current state-of-the-art, but also gave me an experience of a professional scientific research.

Another important factor for this research being of the current quality is the supervision from Diarmuid Kelly, the Co-Founder of BranchKey. He has resulted in a steep enhancement in my learning curve. He has been extremely patient, even with the most elementary questions. He tried to explain stuff in the easiest way possible, and provided relevant resources for me to self study as and when needed. Despite his busy schedule, he was highly available for daily catch-ups, and progress discussions. He created ample opportunities for me to practically test all the implementations or solutions I proposed for the problems at hand. This provided me hands-on experience with production grade systems. In my opinion, such active participation in configuration and maintenance of real world systems not only adds feasibility value to my research, but also prepared me for future industrial experiences.

CONTENTS

1 INTRODUCTION 1
- 1.1 Rationale . . . 2
- 1.2 Research Question . . . 4
- 1.3 Approach and Document Overview . . . 4

2 LITERATURE REVIEW 6
- 2.1 CAP Theorem . . . 6
- 2.2 Databases . . . 7
- 2.3 Distributed Databases . . . 9
- 2.4 Cloud Native Applications . . . 11
- 2.5 Microservice Architecture . . . 11
- 2.6 CAP Theorem in Microservices Architecture . . . 13
- 2.7 Federated Learning Implementations . . . 15
- 2.8 Infrastructure Evaluation . . . 16

3 BRANCHKEY ARCHITECTURE 18
- 3.1 Authenticator . . . 19
  - 3.1.1 Functional Requirements . . . 19
  - 3.1.2 Non-Functional Requirements . . . 19
- 3.2 File Uploader . . . 20
  - 3.2.1 Functional Requirements . . . 20
  - 3.2.2 Non-Functional Requirements . . . 20
- 3.3 File Downloader . . . 21
  - 3.3.1 Functional Requirements . . . 21
  - 3.3.2 Non-Functional Requirements . . . 22

4 EXPERIMENTAL IMPLEMENTATION 23
- 4.1 Authenticator . . . 26
  - 4.1.1 Experiment 1: Master/slave DB distribution . . . 28
  - 4.1.2 Experiment 2: Server side caching . . . 30
- 4.2 File Uploader . . . 30
  - 4.2.1 Experiment 1: Redis Master/Slave . . . 33
  - 4.2.2 Experiment 2: Redis Cluster . . . 34
- 4.3 File Downloader . . . 34
  - 4.3.1 Experiment 1: Data Replication . . . 37
  - 4.3.2 Experiment 2: Data partitioning . . . 38

5 EVALUATION 39
- 5.1 Authenticator . . . 40
  - 5.1.1 Current System . . . 41
  - 5.1.2 Results from the experiments . . . 43
  - 5.1.3 Evaluate the results and compare across relevant parameters . . . 49
- 5.2 File Uploader . . . 50
  - 5.2.1 Current System . . . 51
  - 5.2.2 Results from the experiments . . . 53
  - 5.2.3 Evaluate the results and compare across relevant parameters . . . 60
- 5.3 File Downloader . . . 62
  - 5.3.1 Current System . . . 62
  - 5.3.2 Results from each experiments . . . 64
  - 5.3.3 Evaluate the results and compare across relevant parameters . . . 69

5.4 Discussion 71
6 conclusion and future work 73
6.1 Conclusion 73
6.2 Future Work 75

i appendix 76
a requirement analysis of other branchkey microservices 77
a.1 Authoriser 77
a.1.1 Functional Requirements 77
a.1.2 Non-Functional Requirements 77
a.2 API Gateway 78
a.2.1 Functional Requirements 78
a.2.2 Non-Functional Requirements 78
a.3 Aggregation Task Creator 79
a.3.1 Functional Requirements 79
a.3.2 Non-Functional Requirements 79
a.4 Central Aggregator 79
a.4.1 Functional Requirements 80
a.4.2 Non-Functional Requirements 80

bibliography 81

LIST OF FIGURES

Figure 1 CAP Theorem Triangle and use case placement . . . 3
Figure 2 Current architecture of BranchKey data flow pipeline 18
Figure 3 Authenticator: Service Interface 19
Figure 4 File Uploader: Service Interface 20
Figure 5 File Downloader: Service Interface 21
Figure 6 Research Framework 23
Figure 7 Experimentation Protocol 24
Figure 8 Monitoring Stack 39
Figure 9 Authenticator Dashboard [Current System] 41
Figure 10 Authenticator PostgreSQL Dashboard [Current System] 41
Figure 11 Authenticator PostgreSQL Dashboard Continued [Current System] 42
Figure 12 Authenticator Service Pod CPU Load [Current System] 42
Figure 13 Authenticator PostgreSQL Pod CPU Load [Current System] 42
Figure 14 Authenticator Service Pod Memory Usage [Current System] 43
Figure 15 Authenticator PostgreSQL Pod Memory Usage [Current System] 43
Figure 16 Authenticator PostgreSQL Persistent Volume Usage [Current System] 43
Figure 17 Authenticator Dashboard [Experiment 1] 44
Figure 18 Authenticator Read Service Pod CPU Load [Experiment 1] 45
Figure 19 Authenticator Write Service Pod CPU Load [Experiment 1] 45
Figure 20 Authenticator Master DB Pod CPU Load [Experiment 1] 45
Figure 21 Authenticator Slave DB Pod CPU Load [Experiment 1] 45
Figure 22 Authenticator Read Service Pod Memory Usage [Experiment 1] 46
Figure 23 Authenticator Write Service Pod Memory Usage [Experiment 1] 46
Figure 24 Authenticator Master DB Pod Memory Usage [Experiment 1] 46
Figure 25 Authenticator Slave DB Pod Memory Usage [Experiment 1] 46
Figure 26 Authenticator Master DB Persistent Volume Usage [Experiment 1] 46
Figure 27 Authenticator Slave DB Persistent Volume Usage [Experiment 1] 46
Figure 28 Authenticator Dashboard [Experiment 2] 47
Figure 29 Authenticator Redis Dashboard [Experiment 2] 47
Figure 30 Authenticator Service Pod CPU Load [Experiment 2] 48
Figure 31 Authenticator DB Pod CPU Load [Experiment 2] . . . 48
Figure 32 Authenticator Redis Pod CPU Load [Experiment 2] . 48
Figure 33 Authenticator Service Pod Memory Usage [Experiment 2] 48

Figure 34 Authenticator DB Pod Memory Usage [Experiment 2] 48
Figure 35 Authenticator Redis Pod Memory Usage [Experiment 2] 49
Figure 36 Authenticator DB Persistent Volume Usage [Experiment 2] 49
Figure 37 File Uploader Dashboard [Current System] 51
Figure 38 File Uploader Redis Dashboard [Current System] . . 51
Figure 39 File Uploader Service Pod CPU Load [Current System] 52
Figure 40 File Uploader Redis Pod CPU Load [Current System] 52
Figure 41 File Uploader Service Pod Memory Usage [Current System] 53
Figure 42 File Uploader Redis Pod Memory Usage [Current System] 53
Figure 43 File Uploader Dashboard [Experiment 1] 54
Figure 44 File Uploader Redis Dashboard [Experiment 1] 54
Figure 45 File Uploader Redis Dashboard (Continued) [Experiment 1] 54
Figure 46 File Uploader Service Pod CPU Load [Experiment 1] 55
Figure 47 File Uploader Redis Master Pod CPU Load [Experiment 1] 55
Figure 48 File Uploader Redis Slave Pod CPU Load [Experiment 1] 56
Figure 49 File Uploader Service Pod Memory Usage [Experiment 1] 56
Figure 50 File Uploader Redis Master Pod Memory Usage [Experiment 1] 56
Figure 51 File Uploader Redis Persistent Volume Usage [Experiment 1] 56
Figure 52 File Uploader Dashboard [Experiment 2] 57
Figure 53 File Uploader Redis Master Dashboard [Experiment 2] 57
Figure 54 File Uploader Redis Slave Dashboard [Experiment 2] 57
Figure 55 File Uploader Service CPU Load [Experiment 2] . . . 58
Figure 56 File Uploader Redis Master CPU Load [Experiment 2] 59
Figure 57 File Uploader Redis Slave CPU Load [Experiment 2] 59
Figure 58 File Uploader Service Pod Memory Usage [Experiment 2] 59
Figure 59 File Uploader Redis Master Pod Memory Usage [Experiment 2] 59
Figure 60 File Uploader Redis Slave Pod Memory Usage [Experiment 2] 59
Figure 61 File Uploader Redis Master Persistent Volume Usage [Experiment 2] 60
Figure 62 File Uploader Redis Slave Persistent Volume Usage [Experiment 2] 60
Figure 63 File Downloader Dashboard [Current System] 62
Figure 64 File Downloader PostgreSQL Dashboard [Current System] 63
Figure 65 File Downloader PostgreSQL Dashboard (Continued) [Current System] 63
Figure 66 File Downloader Service Pod CPU Load [Current System] 63
Figure 67 File Downloader DB Pod CPU Load [Current System] 64

Figure 68 File Downloader Service Pod Memory Usage [Current System] . . . 64
Figure 69 File Downloader DB Pod Memory Usage [Current System] . . . 64
Figure 70 File Downloader DB Persistent Volume Usage [Current System] . . . 64
Figure 71 File Downloader Dashboard [Experiment 1] . . . 65
Figure 72 File Downloader PostgreSQL Dashboard [Experiment 1] . . . 65
Figure 73 File Downloader PostgreSQL Dashboard (Continued) [Experiment 1] . . . 66
Figure 74 File Downloader Service CPU Load [Experiment 1] . 66
Figure 75 File Downloader Master DB CPU Load [Experiment 1] 66
Figure 76 File Downloader Slave DB CPU Load [Experiment 1] 67
Figure 77 File Downloader Service Memory Usage [Experiment 1] . . . 67
Figure 78 File Downloader Master DB Memory Usage [Experiment 1] . . . 67
Figure 79 File Downloader Slave DB Memory Usage [Experiment 1] . . . 67
Figure 80 File Downloader Master DB Persistent Volume Usage [Experiment 1] . . . 67
Figure 81 File Downloader Slave DB Persistent Volume Usage [Experiment 1] . . . 67
Figure 82 File Downloader Dashboard [Experiment 2] . . . 68
Figure 83 File Downloader Service Pod CPU Load [Experiment 2] . . . 68
Figure 84 File Downloader DB Pod CPU Load [Experiment 2] . 69
Figure 85 File Downloader Service Pod Memory Usage [Experiment 2] . . . 69
Figure 86 File Downloader DB Pod Memory Usage [Experiment 2] . . . 69
Figure 87 File Downloader DB Persistent Volume Memory Usage [Experiment 2] . . . 69
Figure 88 Authoriser: Service Interface . . . 77
Figure 89 API Gateway: Service Interface . . . 78
Figure 90 Aggregation Task Creator: Service Interface . . . 79
Figure 91 Central Aggregator: Service Interface . . . 80

LIST OF TABLES

Table 1 AWS Pricing Model 40
Table 2 Memory Footprint of the Current System 40
Table 3 Authenticator Evaluation: API Response Times and Server Errors 49
Table 4 Authenticator Evaluation: System Health Metrics . . 50
Table 5 Authenticator Evaluation: Persistent Volume Usage . 50
Table 6 File Uploader Evaluation: API Response Times and Server Errors 61
Table 7 File Uploader Evaluation: System Health Metrics . . 61
Table 8 File Uploader Evaluation: Persistent Volume Usage . 61
Table 9 File Downloader Evaluation: API Response Times and Server Errors 70
Table 10 File Downloader Evaluation: System Health Metrics . 70
Table 11 File Downloader Evaluation: Persistent Volume Usage 70

1 INTRODUCTION

CAP Theorem [22] states that a shared data system can have at most two of the three following properties: Consistency, Availability, Partition Tolerance. As defined by Gilbert and Lynch in [45],

- **Consistency** refers to the assurance that a server returns only the appropriate response per request received.
- **Availability** guarantees that every request receives some response from a running server
- **Partition Tolerance** says that a system can be partitioned into different independent nodes. In events where one or more nodes are out of service, the system still responds to every request.

The past few years have seen ample research in this area. Computer scientists have evaluated each of these properties in immense detail, developed the implementation algorithms, and even evaluated the workarounds in most cases. However, such research has either been very generic, or highly use case specific. For instance, research by Gilbert and Lynch [45] presents a general perspective over the CAP Theorem, and its implications over the achievement of certain non-functional requirements, mainly safety and liveness, in an application. Eric Brewer's revisiting of the topic in [23] presents a general overview of the CAP Theorem in context of present day applications, such as its standing against the modern *Atomic, Consistent, Isolated, Durable* (ACID) [49] and *Basically Available, Soft state, Eventually consistent* (BASE) [42] design philosophies for consistency. However, results of such generic studies are difficult to extend down to discrete industrial scenarios, where use case specific investigations need to be performed per use case basis. M. Stonebraker has recorded the effects of the CAP Theorem over databases [103], but it is mostly a compilation of comments and arguments, very specific to the use case under investigation.

Discussing one such research, Simon in [101] analysed that the CAP Theorem leads to a system with only one of the following three characteristics:

- *High consistency and high availability, but no partition tolerance:* The system in this scenario does not define a set behaviour in case one or more of the components fail.
- *High availability and partition tolerance, but no consistency:* Individual nodes are available, even in case of some failures. However, data across nodes is not consistent.
- *High consistency and partition tolerance, but less availability:* It needs constant data consistency across partitions. Thus, in case of a component failure, system can become unavailable.

These characteristics play an important part in designing the architecture for distributed systems. As the authors of [31] discuss, the CAP Theorem not only dominates the architectural design process, but also helps in making informed decisions about which properties can be prioritised while others are

being sacrificed. They also discuss the possibility of arriving at an optimised compromise of consistency or availability (in addition to partition tolerance), rather than giving them up totally. This paves a way for the application of the CAP Theorem in microservices architectures [81]. The microservices architecture paradigm refers to the breaking down of a bigger application into more fine-grained, loosely coupled smaller components, each performing a small part of the bigger application. This enables the development, deployment, scaling, and maintenance of each smaller component independently. It gives us, the software engineers, an opportunity to divide and conquer the requirements of different parts of the system. It allows us to break down a bigger system into smaller subsystems, and choose amongst consistency, availability and partition tolerance for each of them.

We plan to assess the performance of different architectural designs for each of the three scenarios identified by Simon, as discussed above, however confining them to different components of a microservices architecture, and restricting them to the corresponding business requirements.

1.1 RATIONALE

This research will be conducted in collaboration with BranchKey [18]. BranchKey is a federated learning platform which provides distributed machine learning solutions to its customers. In simple terms, BranchKey exposes APIs for multiple user clients to upload their individual models, which are aggregated over the central server, and the result is flowed down to each client. The current system implements a microservices architecture, where the Single Responsibility Principle [75] defines the boundaries of each subsystem. This provides the ability to tackle the functional and non-functional requirements of each subsystem separately. Over the course of developing this high scalalable, robust and resilient system, the software platform engineers at BranchKey stumbled upon the need for data management systems, each with one of the above described requirements:

1. *High consistency and high availability:* Every request coming onto the platform needs to be authenticated against the user credentials. Moreover, it also needs to be authorised for its permission to access the service it requests. This data needs to be highly consistent. For example, if a user updates its credentials, all the upcoming requests need to match the updated values. Additionally, this data demands high availability, for every request which hits the platform shall first be verified via this authentication and authorisation system. However, since user credentials and permissions are not volume intensive, all of it can be stored in a single partition, thus eliminating the need for partition tolerance.

2. *High availability and partition tolerance:* Every time a client sends in its model for higher level aggregation, it needs to be cached for a fast synchronous response. Later, it can be persisted, and used for aggregation. This cache needs to be highly available as it is a part of the synchronous client API. BranchKey accepts client models up to 1 MB per request. With growing requirements to scale, every aggregation can be up to thousands of clients, making this data collectively huge, especially if multiple user aggregations happen concurrently. Thus, it arises an inevitable need to partition tolerance. But since the aggrega-

tion happens asynchronously, this system can afford eventual consistency [115].

3. *High consistency and partition tolerance:* Customers like to access the history of their clients, in terms of files uploaded by each of them, aggregations happened, clients (and their models) which participated in a given aggregation, etc. As explained in the previous use case, this data collectively becomes huge, and demands storage across partitions. Furthermore, it needs to be consistent across all the partitions. However, since accessing history is only an on-demand non-critical feature, it is acceptable to compromise over the availability of this system

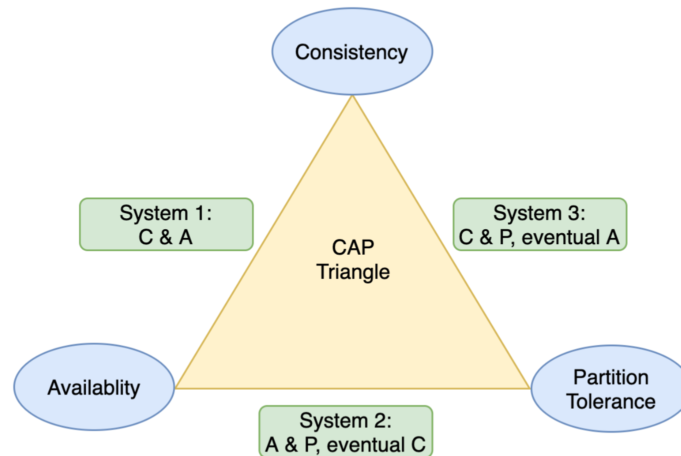


Figure 1: CAP Theorem Triangle and use case placement

This research aims to develop the means to systematically evaluate multiple architectural designs and database configurations for each of the above use cases, to assist BranchKey deploy the best possible strategies, while knowing the trade-offs for each of them. Figure 1 shows the placement of use cases against the CAP theorem limitations in pictorial format.

In a more generic sense, this is an exploratory case study performed along the guidelines provided by Runeson et al. in [95]. The aim here is to investigate different existing architectural designs and patterns for data-focused applications, such as federated learning, that are constrained by one of the three CAP Theorem scenarios, and devise the optimal strategy for the required use case. We will be studying multiple architectural designs, varying from implementing the application level logic for optimising cost and performance while incorporating the CAP theorem constraints, to exporting it down to the database level by using the native database algorithms. Following this, we would weigh the trade-offs between the two solutions for industrial scale deployments, thus providing an experimental study for all the future readers. The broad idea behind this being the establishment of this research as a trustworthy reference for helping application developers identify the optimal architecture per use case basis. In a generic sense, we provide an elaborate and self-sufficient research framework which can be used to evaluate different possible architectural and infrastructural solutions of a data-focused microservice which is impacted by the CAP theorem limitations, and obtain an optimal architectural and/or database solution

for the same, under specified constraints. In specific sense, we apply this research framework to three microservices of the BranchKey platform, and exemplify the process for the same.

In the context of this study, optimal architecture is the one with the least operational expenses and best performance metrics related to the service level objectives. The performance metrics involve minimal response time latency, and error percentages. The operational expenses mainly include the deployment costs over different cloud service providers, along with the network bandwidth usage, the data usage, and their respective costs incurred. It is an accepted industry practice to minimise these costs in accordance with the required performance and scale. This is important to save both, the financial resources as well the development and maintenance effort.

1.2 RESEARCH QUESTION

Formalizing the idea described above, the focus of this research is narrowed down to the following research question:

"How to evaluate different architectural designs towards optimising the overall cost and performance of data-focused microservices affected by the limitations of the CAP Theorem?"

This can be segregated into three sub-questions:

RQ.1 *How do the limitations imposed by the CAP theorem affect the design of a data-focused microservice?*

RQ.2 *How to identify different architectural and infrastructural alternatives for a data-focused microservice with the restrictions imposed by the CAP theorem, while keeping in mind the business requirements of cost and performance?*

RQ.3 *How to decide on the most suitable architectural and infrastructural solution for the optimal cost and performance of the microservice under investigation?*

Important thing to note here is that the cost and performance is analysed in terms described in Section 1.1.

1.3 APPROACH AND DOCUMENT OVERVIEW

This study focuses on evaluating each sub question from Section 1.2 with respect to existing literature and helping the developers identify optimal solutions in accordance with the use cases under investigation. Firstly, we review the existing literature in the concerned fields to examine the effects of CAP theorem restrictions over the choice of architectural and infrastructural designs for data-focused microservices, and narrow down a list of the same to be assessed in the remainder of this research. This review is in Chapter 2. Then, in Chapter 3, we give an overview of the current BranchKey system architecture, and define in brief detail the functional and non-functional requirements of all the modules being investigated in this project. Chapter 4 talks about a generic research framework to use all of this knowledge as an advantageous investigation tool for some BranchKey specific use cases at hand. Using this framework, we design and implement industrial scale experiments for each of the use cases being studied in this investigation.

Chapter 5 provides an evaluation of the results obtained from each experiment in terms of the system's operational cost, computational cost, response latency, fault tolerance, etc. It also discusses the overall viability of the research framework in accordance with the research questions postulated in this chapter. Finally, we draw the conclusions as to which strategies are best suited per use case, and discuss future work in regards to this study in Chapter 6.

2 LITERATURE REVIEW

In this chapter, we extend upon the basic concepts needed for this research, and examine existing literature within similar contexts. This literature review also helps us answer **RQ.1** by analysing different architectural and infrastructural solutions for each of the CAP theorem scenarios. We start by formally defining CAP Theorem, and its alternatives in Section 2.1. We then look through basic properties of different databases in Section 2.2. Section 2.3 combines the idea of CAP Theorem into databases, thus discussing distributed databases. Thereafter, we move on to discuss the implications of CAP Theorem over Cloud Native Applications in general in Section 2.4. Finally we dive into the details of microservice architecture in Section 2.5, and discuss its ramifications with CAP Theorem in Section 2.6. Next, we discuss some practical implementation solutions for federated learning systems in Section 2.7. This is to address specifics of the BranchKey system. Lastly, we discuss ways to evaluate an infrastructural solution in Section 2.8, and use them to compare different selected deployment strategies for a use case at hand.

2.1 CAP THEOREM

A distributed system is a collection of computation units spread over a common network, but working together to provide the facade of a single computer to the end user. These computation units could store multiple partitions of same data, or could individually perform parts of a large computation. This gives rise to the need for three important properties, namely *consistency, availability,* and *partition tolerance* to be analysed deeply while designing such systems. CAP Theorem, as explained in Chapter 1, explains that only two of these three properties can be guaranteed in a distributed system. Gilbert and Lynch define and formally prove this claim in [44]. Kleppmann [58] also provides a detailed explanation of the terms *consistency, availability,* and *partition tolerance*, and their implications over a distributed system. These implications include the consequences of choosing absolutely two of these properties in a system, or sometimes eventually achieving the third property. Extending more into this domain, [78] explores the consistency characteristics of highly available and scalable distributed systems which can tolerate network partitions. It labels *atomic consistency* as the strongest form, and provides a compromise of weakly and eventually consistent systems, along with their formal proofs.

Since CAP Theorem limits the abilities of distributed systems by demanding strict trade-offs, many researchers tried to substantiate middle grounds between these trade-offs. For instance, Martin Kleppmann [58] proposes a delay sensitivity framework for reasoning about trade-offs between consistency guarantees and tolerance of network faults in a replicated database. Similarly, in [2], Abadi says that CAP Theorem only talks about failure cases, where network partitions, and the database has to choose between availability and consistency. It does not talk about normal cases. Failures are rare. And in fact, in normal scenarios, distributed databases are fully capable

of providing ACID transactions. Thus, he proposes PACELC, which considers network latency as an alternative to partition tolerance, which may help practitioners reason about the trade-offs between consistency guarantees and tolerance of network faults, especially in replicated databases.

Continuing over the same line of thought, [65] provides a another alternative to CAP theorem by introducing CAL theorem. It takes into account network latency instead of partition tolerance, and assumes infinite latency to be same as an unreachable network partition. Further, it discusses the trade-offs between consistency and availability, by examining centralised coordination against decentralised coordination. [53] gives a fresh perspective to the idea of consistency in a system. It says that if a problem is monotonic in a coordination-free program, it guarantees consistency in all executions. However, non-monotonic problems require run-time consistency checks. Thus, we, as software engineers, can totally avoid consistency checks in monotonic problems. Likewise, [4] introduces the concept of consistability, a combination metric to measure the consistency and availability of a distributed system in case of failures.

Converging all the theoretical knowledge gathered from the aforementioned articles, we explored some more literature to look for implementation details. [13] describes a middle layer tool to convert weak consistency to strong consistency. Considering the alternatives to CAP theorem, which suggested the substitution of partition tolerance with network communication latencies, the article *CAP Twelve years later: How the "Rules" have Changed* [23] proposes a solution to be able to optimise both consistency and availability when a network partition exists, by taking into account communication latencies. Alternatively, Vogels, in his research *Eventually Consistent* [115] meticulously enlists the situations when we can choose eventual consistency over strong consistency in distributed systems.

2.2 DATABASES

A database (DB) is a data storage provision, organised and optimised for easy access, management, modification and control. Priority order of each of these properties gives rise to different categories of databases. For instance, we have databases which support ACID or BASE transactions, as described in detail by D. Pritchett in [86]. Briefly stating, ACID states for *Atomic, Consistent, Isolated* and *Durable* database transactions, whereas BASE is for *Basically Available, Soft state, Eventually consistent* transactions. Furthermore, databases can be distinguished based on how data is stored. Relational databases (also called SQL databases) store data in the form of tables, with inter-table and intra-table relationships established to optimise access. Non-relational databases (also called NoSQL databases) store data without a fixed schema. These can again be categorised into wide column stores, document store, graph databases, key-value stores, object stores, time series databases, etc. The survey [29] performs a detailed analyses of 15 categories of NoSQL databases, and proposes the principles to choose the best suited NoSQL database for different enterprise use cases. Another study, [57], extends the classification of NoSQL databases by incorporating features such as CAP characteristics, free/proprietary ownership, etc in addition to the data model. This survey, performed over 80 NoSQL databases, also narrows down the best suitable ones for different applications. *Memcached, Redis, and*

Aerospike Key-Value Stores Empirical Comparison [10] performs a similar analysis for key value stores.

Once a competent database is selected for the use case at hand, it can be scaled or performance optimised further via multiple mechanisms. Data partitioning or sharding allows to divide data horizontally or vertically and store it in different tables within a single node, or across different nodes distributed in a network respectively. Data replication facilitates duplicating critical data across multiple copies, which can either be stored on the same node, or on different nodes. Distributed control allows for actual data storage to be separated from the database management system server. Caching enables the server to store a copy of frequently accessed data, so as to serve future queries faster. This can be implemented over any of the previously described mechanisms, i.e., partitions, replications, or even distributed control. These mechanisms demand additional infrastructure in the form of additional nodes in the network, either to store chunks of data, or to serve the queries. Thus, these solutions claim a wise decision to be taken in regards of whether the additional infrastructure cost is worth the additional performance claims. [102] gives a performance comparison between these mechanisms, where as [59] compares them in regards to their infrastructure costs across different cloud providers in the market. Rehmann and Folkerts [92] also provide an extensive performance evaluation mechanism for database deployments in cloud infrastructure.

Moving on to the implementation details of data intensive applications, selecting an adept database and modelling the schema therein are not the easiest things to do. Aforementioned literature does compare multiple databases over certain properties, and gives a framework to make a selection, but it does not describe which properties to look for, when designing for a certain use case. Cattell [105] gives a list of rules to which can be used to design the database requirements for a use case at hand, and thus stipulate the "properties" to look for.

Once this selection is made, schema modelling is the next thing to do. This part of the process is fundamentally based on the type of database, i.e., non-relational or relational. In case of non-relational databases, it becomes tricky for it requires a change in our very basic understanding of data as a tabular entity. Moreover, every non-relational database compels for a different model. For instance, the modelling for a key-value store has to be entirely different from a graph-database. The survey [112] enlists many resources which can be referred to for this. It focuses on *"contexts like benchmark, evaluations, migration, and schema generation, as well as features to be considered for modelling NoSQL databases, such as the number of records by entities, CRUD[1] operations, and system requirements (availability, consistency, or scalability)."*

For relational databases, schema can be designed in coherence with the idea of entity relations in an object oriented application. Data is observed in tabular form, and modelled into database design patterns. These database design patterns can be incorporated into the rudimentary database code, or into the schema designed over it. Merlowe, Ku, and Benham propose a pedagogy to study the rudimentary database design patterns in [73]. However, these are are very difficult to implement, and we mostly scope this project

1 *C*reate, *R*ead, *U*pdate, *D*elete

towards schema level designing in existing databases. *Relational Database Design Patterns* [50] gives an extensive list of 24 such design patterns, with an aim of easing database architecture for industrial applications. Eessaar, [35] proposes a practical implementation of these patterns with a code generating tool which automates the generation of database queries based on the selected database pattern.

Considering some real world examples of implementing the above discussed database mechanisms and patterns, the researchers at Google recount the semantics and the replication algorithm of Megastore in [14]. They claim that Megastore is designed to provide the high availability and horizontal scaling abilities of a non-relational database with the strongly consistent ACID transactions of a relational database. Similarly, [88] demonstrates a rudimentary database design algorithm by optimising query execution for a partitioned database. However, as stated before, such algorithms are very low level, and are beyond the scope of this project.

2.3 DISTRIBUTED DATABASES

As defined in [84], a distributed database is a collection of physically separated chunks of data stored of different nodes connected over a network. These data chunks can be replication copies of same data, or smaller partitions of a bigger data. Since these data chunks are to be made accessible to the end user as a single database without exposing the internal nodal distribution, they are expected to be compliant with CAP theorem requirements as well [104]. As explored by A distributed database can be one of *CA (consistent and available)*, *AP (available and partition tolerant)* or *CP (consistent and partition tolerant)*. Since it is a "distributed" database, it is expected to be partition tolerant. Thus, we mostly examine the trade-offs between consistency and availability in accordance with the specific use case.

Consistency becomes important if none of the users can afford to be served an older version of the data. However, this means that every update needs to be committed and communicated to all the nodes in the network before approving it as a successful update. This is bound to slow down the response time, or even lead to infinite latency (or failure) if network partitions and some nodes become unreachable in the network. Consequently, it lowers the availability quotient of the system. On the contrary, if the requirement is to be available at all times, consistency needs to be discounted. However, this does not mean a completely inconsistent system, rather an eventually or weakly consistent system arises. Different models of consistency, namely weak, PRAM[2], causal, sequential and strict, etc., are available w.r.t. distributed databases. *A brief survey on replica consistency in cloud environments* [24] defines in detail many of these for replicated databases, and also discusses their trade-offs against multiple properties like performance, scalability and latency. A similar study is done by Diogo, et al., [32] where they discuss the consistency models, and their trade-offs with availability in distributed databases which provide partition tolerance by default.

Data can be partitioned vertically or horizontally. Vertical partitioning, as described in [79], refers to breaking down a table along columnar lines to store different data in different tables. This is called normalisation [56] in

2 *P*ipelined *R*andom *A*ccess *M*emory

the world of relational databases. Horizontal partitioning is splitting a table along rows, such that every sub-table contains a subset of all the data rows. In both cases, all of these partitions can either be stored on a single node, or in many different nodes, often distributed physically over a network. The basic idea behind this is to enhance the performance and availability of the database. Horizontal partitioning can be of different types, like range partitioning, schema level partitioning, graph-level partitioning, etc. [15] reviews many of these partitioning techniques from the perspective of reduced distributed transactions and enhanced scalability of distributed databases. If every partition is stored in a different physical node, it is called sharding. [30] explores the idea of sharding, using the technique of hash partitioning. Distributed databases can also be subjected to transaction partitioning, where a query can be served from multiple data warehouses. [16] analyses the effect of workload-driven data partitioning model over the performance and scalability of a distributed database.

Although partitioning eases the access for non-relational databases, it is still a difficult paradigm for relational databases. [3] evaluates the best strategies to horizontally partition a relational database, while keeping in mind the tabular relations, indexes and the joins.

Data replication is another widely used strategy to amplify the security, availability, bandwidth, reliability, and lower response time of data access in both relational and non-relational databases. [100] presents a comprehensive aggregation of many data replication schemes categorised as deduplication, auditing or handling, and relatively compares them. Replication also strongly impacts the consistency and availability of a distributed database. Intuitively, higher replication requirements would add to consistency as well as availability, if only replication happens asynchronously. Synchronous replication might end up reducing response times, but increase reliability and availability in times of partial node failures. Continuing the investigation in this field, [40] discusses the impact of data replication on different consistency models (strong and weak consistencies), as well as propose replication techniques for highly available systems.

Strong consistency, in general, is a difficult notion to achieve in distributed systems. It requires every node to be in constant communication with every other node, thus making the system unavailable in times of network failures. Bailis et al. study this in [12]. They discuss the downsides of strong consistency in distributed systems, and suggest ways to implement eventual consistency instead. Google's in-house distributed storage system for structured data, BigTable [28] is an example of one such system which claims to be a highly available and partition tolerant distributed system with eventual consistency. Saito and Shapiro [96] also provide practical implementations of asynchronous data replication (as discussed above), which *"propagates changes in the background, discovers conflicts after they happen, and reaches agreement on the final contents incrementally."* Theoretically, this choice of eventual or weak consistency should lead to relatively higher availability, lower latency and better network partition tolerance. [116] investigates whether that is actually the case in practical scenarios or not. It studies the consumer level observation of consistency and performance in cases of both, the strongly consistent and the weakly consistent system configurations offered by different platforms, and infers that it can rarely be distinguished in non-failure

scenarios. The researchers in [116] conclude that taking up weaker consistency configurations rarely add up any extra benefit to the system.

Additionally, there are some use cases which strictly demand strong consistency over a distributed system. For example, a financial transaction database demands strong consistency to avoid any monetary discrepancies, but also high availability and partition tolerance for better consumer experience. [62] proposes an architecture design for such a system in a distributed NoSQL database. Similarly, [119] presents a transaction manager for a scalable distributed database, which not only guarantees high availability, but also ACID transactions over multi-node queries even in times of network partition or server failure. Such configurations are not organic to the nature of distributed databases, but definitely offer a solution in cases where any compromise on CAP theorem is unacceptable.

2.4 CLOUD NATIVE APPLICATIONS

Cloud native applications is an organised assembly of multiple independent services, coupled loosely and deployed continuously, to exploit the benefits of cloud computing architecture. A survey by Kratzke and Quint [61] discusses the concepts of cloud native applications in detail, and extensively enlists the literature for research in the domain.

In this project, we restrict the study with regards to the interference of CAP Theorem and cloud native applications. Andrikopoulos, Fehling and Leymann in [8] researched the ramifications of application design paradigms over the CAP attributes of cloud native applications. Further, Andrikopoulos in [9] continued the research in this domain, by proposing an application design methodology for cloud native applications, by taking into account the required CAP theorem trade-offs. This research provides an easy to use framework which facilitates the estimation of CAP theorem implications on the cloud native application under investigation, and makes provisions to adapt the design accordingly.

Providing pragmatic implementation details in this regard, [54] provides an extensive list of 24 design patterns for cloud applications. Each of these patterns is classified according to the quality attributes they satisfy for their major functionality, as explained in the provided sample code implementations. It also discusses the implications of data consistency policies over data intensive cloud applications.

2.5 MICROSERVICE ARCHITECTURE

Speaking informally, the development of a software application has aged from monolith architecture to service oriented architecture (SOA) to microservices architecture in use these days. The main idea behind microservices is that every individual microservice is individually responsible only for a small portion of the entire application. This leads to dealing with scalability, maintainability, flexibility, reliability, security, performance and availability of every microservice individually, thus eliminating the need to tackle these non-functional requirements on a massive scale of the entire application at once. [34] discusses these aspects in more details. Krylovskiy, Jahn and Patti in [63] also briefly argue towards the eminence of microservice architecture in comparison to other prevalent ones, especially in regards

to acceptance for technological heterogeneity, resilience, scaling, organizational alignment and composability. Extending the applications of microservices architecture to other aspects, [6] vouches for using similar organisational principles into system development as a whole. They claim that this is better than service oriented architecture, majorly in terms of *smart endpoints* used by SOA to communicate amongst services, as opposed to *dumb pipelines* used by microservice approach. Augmenting along the same lines of thought, Zimmermann presents an elaborate comparison of the microservices architecture against the service oriented architecture in [123].

As much as theory suggests that microservice architecture enhances the quality attributes of a software application, defiant beliefs demand practical proofs. Hasselbring and Steinacker performed an empirical analysis over the scalability, agility and reliability of software systems in the German e-commerce website otto.de. In [52], they discuss how microservice architecture helped them in terms of vertical decomposition of the application, achieve loose coupling and eventual consistency amongst modules, enhanced scalability and fault tolerance, and eased deployment and development. Identically, the researchers at one of the biggest technical companies, Accenture and Infosys, collaborated in [98]. They have pragmatically shown that migrating an industrial scale banking application from monolith to microservice architecture allowed for a reduction in system application faults by facilitating the developers to localise incoming faults, and reduce the fixing, building and deploying effort.

Since microservice architecture is a collection of small loosely coupled modules, they need to be assembled and interacted with in certain mechanisms based on specific requirements. This gives space for architecture design patterns amongst these modules. [83] gives an extensive list of many architectural patterns for implementing and optimising microservices architecture. Taibi, Lenarduzzi and Pahl also examined several microservices architectural patterns in [107], and summarised their advantages and disadvantages based on certain case studies. This evaluation is mainly in terms of quality attributes every pattern promises to deliver. Reversing the perspective, there can be scenarios where we need to engineer certain quality attributes. To realise the patterns suitable for each of these cases, [111] provides a mapping between quality attributes and architectural patterns.

This project is focused over data intensive use cases. Owing to this requirement, we looked into architectural patterns specific to database applications in microservice domain. Messina et al. introduce the *Database-is-the-service* pattern in their articles *A Simplified Database Pattern for the Microservice Architecture* [77], and *The Database-is-the-Service Pattern for Microservice Architectures* [76]. In layman's words, it transfers the responsibility of the database management system and the corresponding data storage from the owner/prime user of the data to an external cloud architecture provider. Note that this is different from the *Database-per-service* pattern, where every service owns and manages its database. Database-is-the-service does lighten up the load of application, but adds external dependencies and reduces control for the data owner. In [121], Xi analyses the pros and cons of this pattern. While these articles strongly advise splitting of the database across microservices in either database-is-the-service fashion, or the database-per-service fashion, they do not provide the necessary guidelines. Kholy and Fatatry researched this aspect of the problem, and proposed a framework for

managing the database split across microservices, while maintaining data consistency and low latency. In [37], they follow the database-per-service approach, by performing the split corresponding to the business boundaries of each microservice.

Gathering from all the aforementioned literature, as much important as it seems to split a database, there are also downsides of not doing it. [110] clearly states as to how a shared database becomes an anti-pattern in a microservice architecture. In fact, [120] gives an in-depth analysis of real-world industrial implementations of microservice architecture, and the implications of not following the prescribed architectural conventions like decentralised data management, infrastructure automation, architectural patterns, monitoring strategies, etc. As defined in [41], such *"symptoms of bad code or design that can cause different quality problems, such as faults, technical debt, or difficulties with maintenance and evolution"* are called architectural smells. To give better understanding of such pitfalls, and to be able to avoid them when possible, [80] systematically reviews existing literature around the design conventions and architectural smells for microservices. It also proposes ways to refactor existing smells in the codebase.

The idea of many microservices working in coherence with each other to serve user queries also coincides with the concept of a distributed system. Thus, the intersection of these two domains of computer science is inevitable. [97] evaluates the implementation of distributed systems with respect to client-server paradigm, service-oriented architecture, and microservices architecture. They provide a feature analysis of distributed systems against each of these architectures, and conclude that microservices is the best suitable one.

2.6 CAP THEOREM IN MICROSERVICES ARCHITECTURE

Combining microservices with distributed systems gives rise to an entirely new arena of fault tolerance and CAP requirements in microservices. Gill and Buyya discuss many fault tolerance pitfalls for distributed systems with microservices architecture in cloud infrastructure, and propose solutions for some industrial use cases in their research [46]. Making this discussion generic, [51] proposes decision models to take into account the implications of availability and consistency while designing microservices. It exemplifies many patterns like service discovery, versioning, service registration, caching, and load balancing. Narrowing down the focus to database intensive microservices, [94] implements master-slave replication for SQL databases and eventual consistency for NoSQL databases in microservices ecosystem.

As described above, the overlap of CAP theorem and microservices architecture is ineluctable. Thus, it has led to many software scientists discussing this in their studies. Sometimes they just brush past the subject, while other times they discuss it at required lengths. For instance, [26], while comparing SOA and microservices architecture, and mapping of the qualities of microservices to their evident studies, minutely mentions CAP requirements as a part of fault tolerance aspect. Similarly, [25] references the assumed ignorance of CAP theorem while migrating from monolith to microservices architecture. Brown and Woolf [25] also discuss the implications of CAP the-

orem over the architectural patterns for microservices, especially a key-value store.

Alternatively, some studies even propose an alternative for CAP theorem in microservices architecture. One such substitute is *BAC - Backup, Availability, and/or Consistency* theorem proposed by Pardon, Pautasso and Zimmermann in [85]. They consider the requirement of being able to back up a microservice in events of failure as the *partition tolerance* equivalent of CAP theorem. They claim that in an event of failure, should a entire application designed using microservices architectures needs a backup, both availability and consistency cannot be provided. We can only opt for either of them. They prove this by considering the cases of *consistent backups with limited availability* and *eventual inconsistency with full availability*.

The two scenarios described by BAC theorem can be compared to the *CP* and *AP* from the CAP theorem. Exploring both of them in detail, firstly lets discuss *CP*. For data intensive microservices, *CP* refers to strong data consistency. [39] provides an in-depth analysis of the three state-of-the-art data consistency models in microservices architecture, namely, strong consistency, weak consistency and final consistency. It compares these with ACID transactions, and propose ways to achieve each of them. [68] also discusses the ways and the implications of implementing data consistency in microservices architecture. It mainly explores private database per microservice, and cloned database across all microservices.

The alternate scenario is *AP*. This demands for higher availability, and allows for eventual consistency. Contrasting to Section 2.3, satisfying these requirements on architectural level allow for domain level implementations of eventual consistency and/or high availability. Braun, Bieniusa and Elberzhager [19], advocate that choosing eventual consistency over strong consistency shifts the responsibility of data consistency from the infrastructure to the application domain. Further, they also propose ways to implement eventual consistency in the domain layer. Another article [43] elaborates over the three main challenges of migrating a monolith to microservices, i.e., multitenancy, statefulness and data consistency. While discussing ways to resolve the third one of these, they perform an intricate comparison of a weakly consistent *AP* system against a strongly consistent system leading to lower availability, and conclude with preferring the former over the latter. Further, they also discuss ways to implement eventual data consistency in the domain layer.

Moving on to the practical aspects of implementing eventual consistency in the domain layer of a microservices architecture, Braun and Deßloch present ECD3 framework to design eventually consistent domain driven models based on data replication semantics [20]. Adapting these theoretical concepts into real world applications, [33] describes the monolith to microservices migration of the core mission critical system of the Danske Bank. They acknowledge the implications of CAP theorem, and choose to design a highly available and partition tolerant system which offers weak consistency, to ensure low latencies and better consumer experience even in times of network partitions. Following the idea of exploring the pragmatic problems faced by engineers working towards implementing microservices, the researchers of [21] conducted a close observation research over the team of

a multi-site platform development project. They studied the practical implications and problems in designing distributed data intensive systems, and recommended eventual consistency as the safest way to move forward.

An *AP* system not only requires eventual consistency, but also needs a high availability. Availability, as opposed to eventual consistency, is easier solved on infrastructure level than on domain layer. [99] evaluates synchronous (REST[3], gRPC[4]) and asynchronous (AMQP[5] using RabbitMQ) interprocess communication methods in microservices, only to conclude that asynchronous communication delivers higher availability, especially as the number of users in the system increase. Márquez and Astudillo [74], reveal many architectural patterns which can be implemented to enhance the availability of a microservices based application. They highlight the importance of a circuit breaker for synchronous communication, service registry for decoupling the physical address from registered address, and that of asynchronous messaging for reliable communication. However, implementing all of these patterns individually adds to the development overhead. [113] introduces a tool which abstracts away all of this functionality, and reduces the development overhead. It is an alternative to data redundancy strategies like data replication over multiple places, or pub-sub communication across multiple microservices. It rather makes the database of one service available with a read-only view to all the other services, thus reducing the need for inter-service communications to *fetch data*.

2.7 FEDERATED LEARNING IMPLEMENTATIONS

As explained in Section 1.1, BranchKey is a *federated learning as a service* platform. This is relatively a new area of software development, with comparatively less industrial scale implementations. Thus, the peculiar pitfalls of implementing this system for production usages are still being explored. There are many existing implementations with modular architecture. For instance, [71] presents a Python library which can be integrated easily into a federated learning system at client as well as server levels. Similarly, [60] proposes *"Federated Learning as a Service system enabling 3rd-party applications to build collaborative, decentralized, privacy-preserving ML models"*. This implementation is also follows modular service architecture, and evaluates the solution in terms od memory usage, computation costs and on-device power costs. Further, [122] proposes an easy to use beginners' tool to implement and experiment with federated learning. It also exposes its internal architecture, which promises a modular system with efficient deployment. While these systems guarantee easy implementation and prototyping, [17] claims to be the first production grade federated learning system for running on mobile phones. It uses actor design pattern with the tensorflow library.

Moving into the domain of serverless systems with cloud deployments, [48] presents a framework for serverless federated learning *Function-As-A-Service* (FaaS), which can be hosted on any cloud platform, an on premise data center or even on edge computing devices. A similar FaaS implementation is also presented by [27]. While FaaS interfaces are easy to use, they limit control over multiple aspects of the system. [118] instead, exemplifies a python based implementation of federated learning using serverless AWS

3 *RE*presentational *S*tate *T*ransfer
4 *G*oogle *R*emote *P*rocedure *C*all
5 *A*dvanced *M*essage *Q*ueuing *P*rotocol

Lambda service. This explains the development of a production grade system from scratch, and provides a detailed evaluation of infrastructural costs of its deployment in AWS Lambda. [36] presents another open source tool to implement and integrate federated learning into a consumer software. It ensures scalable, efficient and robust federated learning. It even explains the intern system architecture and the programming patterns followed.

As promising all these tools and frameworks sound, they are very specific to their own implementations. The architectural patterns they use, or the infrastructural solutions they implement are not generic in a sense to be blindly adopted by BranchKey. To answer for such questions of generic architectural and infrastructural solutions for federated learning systems, [70] compiles a list of architectural patterns for designing federated learning systems. [70] not only categorizes them per use case, i.e., client management, model management model training and model aggregation, but also evaluates them with detailed benefits and drawbacks. Further, [69] surveys the state of the art for federated learning in software engineering perspective including architecture design, implementation and evaluation.

2.8 INFRASTRUCTURE EVALUATION

The third research question **RQ.3** in Section 1.2 imposes the need for tools to evaluate and compare different infrastructural solutions against the required specifications. These comparisons can happen via graphical representation of metrics data collected from the system, or via absolute parameter values observed. [66] presents *CostHat*, an approach which can help to optimise a microservice architecture by modelling its deployment costs, computation costs, and even network bandwidth costs caused by IO. It can even be integrated into the developers' Integrated Development Environment to raise alerts about potentially costly code changes. The main essence of *CostHat* is to predict the costs based on certain pre-devised heuristics. The results can differ from the real observations under many probabilities. Timon Back, in his study [11] rather presented a simulation based approach, where a simulator *"assists in finding the Pareto optimal hybrid deployment strategy for cost minimization of any cloud application, in both low load as well as bursty load scenarios"*. Extending this, Reuter et al [93] provide a simulation tool for deciding on a cost optimal deployment solution for a hybrid serverless and serverful application.

While simulation and prediction based tools do promise efficient results, they are still bound to overlook certain corner cases. The most efficient calculation can only be done at run-time with production load. Although such run-time monitoring tools reduce the space for action and rather demand immediate fixing as per need, they give the most reliable results. Patrick Vogel [114] proposes one such model to monitor the deployment costs and resource wastage for a containerised cloud based application at run time.

[5] presents an approach to consider the deployment cost of an application over multiple cloud service providers, and recommend an orchestration plan for the application deployment, not only in agreement with cost constraints, but also envisioning the scalability needed for peak load times. As far as an actual tool to implement such a plan is concerned, [72] presents *"a microservices elastic management system for cost reduction in the cloud, which is designed to optimize usage of a single cloud-server instance before it needs to scale*

out to a second instance". It mainly aims to provide a reliable cost efficient solution without using the overpriced cloud-provider services like dynamic load balancing and native cloud autoscaling.

One of the most important contributors to the operational cost of a software application is the infrastructure deployment cost. Since BranchKey is far from setting up their own cloud engine, they are bound to use the services offered by one of the key cloud service providers in the market. However, making a selection out of the available choices in this aspect needs to be a well informed decision. [67] presents a tool to compare different cloud providers in terms of performance and cost efficiency. It mainly monitors the elastic computing, persistence storage and networking services offered therein. Simultaneously, [55] proposes a stochastic metric approach to analyse and quantify the availability of a cloud service, by taking into account their geographically distributed data centers. Diving into more specific analysis, [117] provides an extensive comparison of Google App Engine with Microsoft Azure across multiple aspects like portability, and ease of use. In context of our project, it provides a basic comparison approach in which we can also observe the currently used cloud provider in BranchKey, i.e., AWS.

3 BRANCHKEY ARCHITECTURE

As described in Section 1.1, BranchKey [18] is a Federated-Learning-As-A-Service platform. It allows users to communally train, and devise the learnings of a centrally shared prediction model, without sharing their individual training datasets with other users in the network. This dissociates the conjoint relation between the need to share and store data over cloud, and the ability to perform prediction across a distributed multi-user system. In short, it is distributed machine learning which trains an aggregated model using segregated data. However, just the algorithm alone is not enough to make this application consumer-friendly. It needs a whole data flow pipeline, which ensures security, reliability, scalability, maintainability and desired performance. In this regard, BranchKey follows a microservices architecture with the design shown in Figure 2.

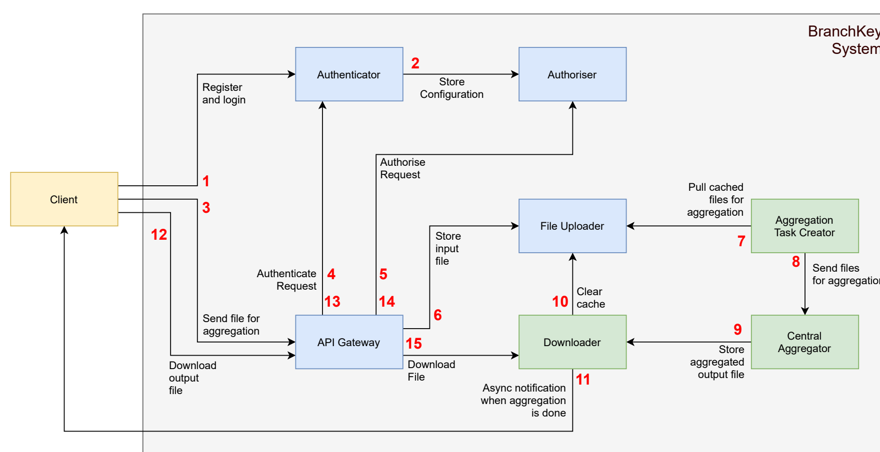


Figure 2: Current architecture of BranchKey data flow pipeline

Here, each box shows *one* microservice. Services which fall into the synchronous flow from the client are shown in *blue*, while the ones which work asynchronously are shown in *green*. *Gray* box encloses the BranchKey system. External client, shown in *yellow*, refers to the end user. Numbers in *red* define the sequence of steps performed across the whole pipeline, starting from registering a user to triggering an aggregation to downloading the aggregated results. Important to note in the figure is that an arrow from *A* to *B* means that the corresponding action is initiated by entity *A*, and is answered upon by entity *B*. Step number is written next to the initiator of the respective step in the pipeline.

Of these all, we will focus our investigation only on three components, namely, `authenticator`, `file uploader` and `file downloader`. Each of these is a data intensive microservice, falling on a different edge of the CAP triangle, as shown in Figure 1. As discussed in Section 1.1, `authenticator` is a *CA* system, `file uploader` is a *AP* system, and `file downloader` is a *CP* system. Let us look into the details of each of these microservices[1]. For each

1 Requirement analysis of all the other microservices is provided in Appendix A

of them, we will examine their communication interface for other services in the system, their dependency graph, and their functional and non-functional requirements.

3.1 AUTHENTICATOR

`Authenticator` is responsible for user data and their authentication into the system. As shown in Figure 3, it allows to create a new user, and stores its credentials for facilitating login and logout. Additionally, it also provisions the authentication check for users whenever they make a request into the system by examining if they are allowed accesses within the system.

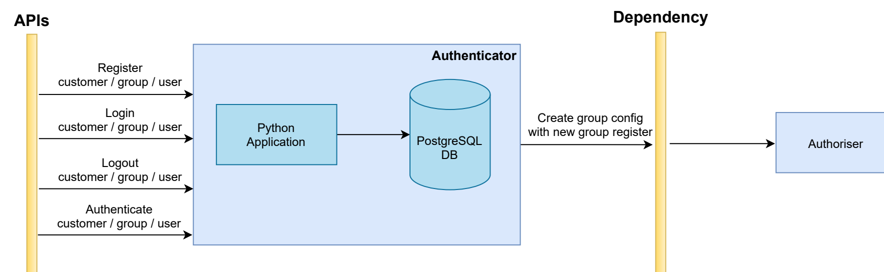


Figure 3: Authenticator: Service Interface

3.1.1 *Functional Requirements*

Following are the functional requirements for this system:

- Should allow user creation in context of their respective roles: customer / group / user
- Should store all user details
- Should create authorisation configuration on new group registration
- Should not allow a customer to create a group under other customers
- Should not allow a group to create user under other groups/customers
- Should allow session management and authentication

3.1.2 *Non-Functional Requirements*

Following are the non-functional requirements for this system:

- Should be able to scale for different APIs accordingly:
  - Write APIs: Create, Login and Logout User: Less frequent: total of 25 *requests per second*
  - Read APIs: Authenticate User: Very high frequency (25 *requests per second*), called from API Gateway for any request received
- All APIs should respond within *200 ms*.
- Strong consistency over all data
- Highly Available, especially for read APIs

- Create and Login APIs are external: should have basic security mechanisms
- Should push application logs and metrics to appropriate end points

3.2 FILE UPLOADER

`File-uploader` temporarily caches the user data file, and its corresponding user information before aggregation with the data from other users. Once the aggregation is completed and results are permanently stored, it expects this cache to be cleared. As shown in Figure 4, it exposes the two described enpoints, and has no downstream dependency.

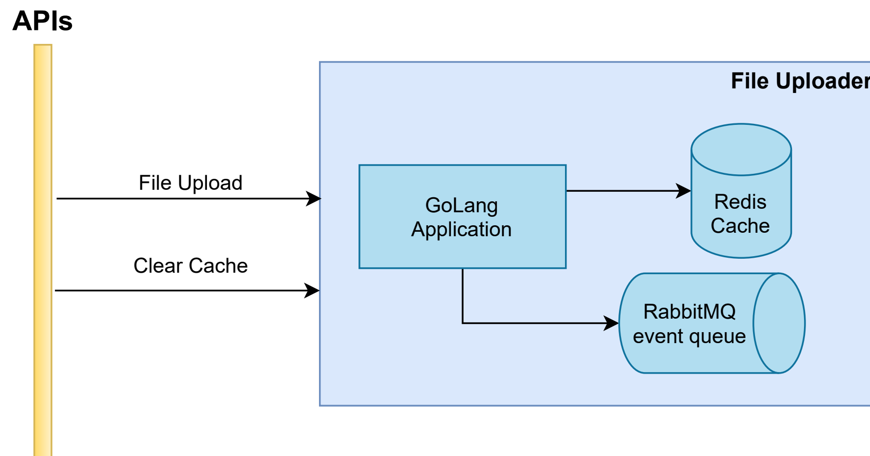


Figure 4: File Uploader: Service Interface

3.2.1 *Functional Requirements*

Following are the functional requirements for this system:

- Should cache the input file uploaded by the client, until the corresponding aggregation is complete
- Should store the related index data (owner-id, group-id, group-config, file-metadata, etc) for each of these input files
- Should expose an endpoint to clear this cache once aggregation is successfully completed

3.2.2 *Non-Functional Requirements*

Following are the non-functional requirements for this system:

- Should scale to receive input files from all the clients at *20 requests per second*
- Should scale to handle a file size of upto *1 MB* per request[2]

2 It is a 3 layer convolutional neural network with shapes `[1 x 32]`, `[32 x 32]` and `[32 x 64]` connected to a fully connected Rectifier Linear Unit of dimension `[(3 x 3 x 64) x 256]`. Thus a total of `3 x 3 x 64 x 256 = 1,47,456` variables. Since each variable is a *float* value, it takes 8 bytes to store each value. Hence, we need a total of `1,47,456 x 8 = 11,79,648` bytes or `1.18 MB`.

- Scaling might need deployment across multiple network partitions like different data center availability zones, regions, etc. Thus, should highly prioritise availability and tolerance to network partitioning
- Should be eventually consistent (strong consistency not needed) because further steps down this pipeline happen asynchronously
- Should have as low response time needs as possible (up to a maximum of *400 ms*), because this is a part of sync call from client
- Should push application logs and metrics to appropriate end points

3.3 FILE DOWNLOADER

`File Downloader` stores the aggregated output result file received from `central-aggregator`, and clears the cache in the `file-uploader`. It then informs the client when an aggregated output file is ready for download. This client intimation happens via an event queue. It also provides the download API for the client to download the corresponding aggregated output file. The exposed endpoints and external dependencies of `file-downloader` are summarised in Figure 5.

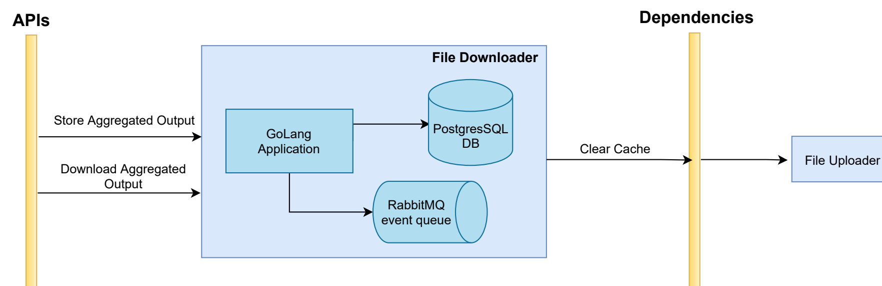


Figure 5: File Downloader: Service Interface

3.3.1 *Functional Requirements*

The functional requirements for `file-downloader` are as follows:

- Should expose an endpoint to upload aggregation output result file
- Should store a list of all the output files, along with their group-ids and group-configs
- Should send out asynchronous events to clients once this output file is ready to be downloaded
- Should expose an endpoint that allows clients to download these output files
- Should verify if the calling client is authorised to download the requested file
- Should delete the output files after their corresponding expiry time set in group-config

3.3.2 *Non-Functional Requirements*

The non-functional requirements for `file-downloader` are as follows:

- Should be scaled enough to store all the output files as per need
- Should be able to respond on all the endpoints within reasonable response time
- Should be scaled differently for each API:
  - Write: file upload would be called only on the completion of each aggregation, i.e., it would have less throughput
  - Read: file download would be called at least once by each client participating in every aggregation, i.e., it is expected to have relatively higher throughput
- Since this serves history, which is a non-critical on-demand feature, it is acceptable to have less availability for this system
- Should have high consistency, so that clients get the correct output file
- Should persist files, at least until their expiry time
- Should push application logs and metrics to appropriate end points

4 EXPERIMENTAL IMPLEMENTATION

In layman's terms, the aim of this thesis is to enable the enhancement of an existing microservice, or even design it from scratch, in compliance with its functional and non-functional requirements, constrained by CAP theorem limitations. As an exemplary use case, we will perform this examination over BranchKey microservices. The idea is to evaluate their current implementations against their stipulated non-functional requirements in terms of consistency, availability and partition tolerance, and propose enhancements if needed. In pragmatic terms, we hope to present a general procedure for a software engineer to evaluate any microservice under such criteria, and decide on the most suitable architectural and infrastructural solution for the same. To facilitate this, we present a generic `Research Framework` (RF), which will enable the aforementioned evaluations.

Figure 6: Research Framework

The proposed research framework, shown in Figure 6, is a series of seven steps. However, the sixth step is further decomposed into seven sub-steps again, to form the `Experimentation Protocol` (EP) shown in Figure 7.

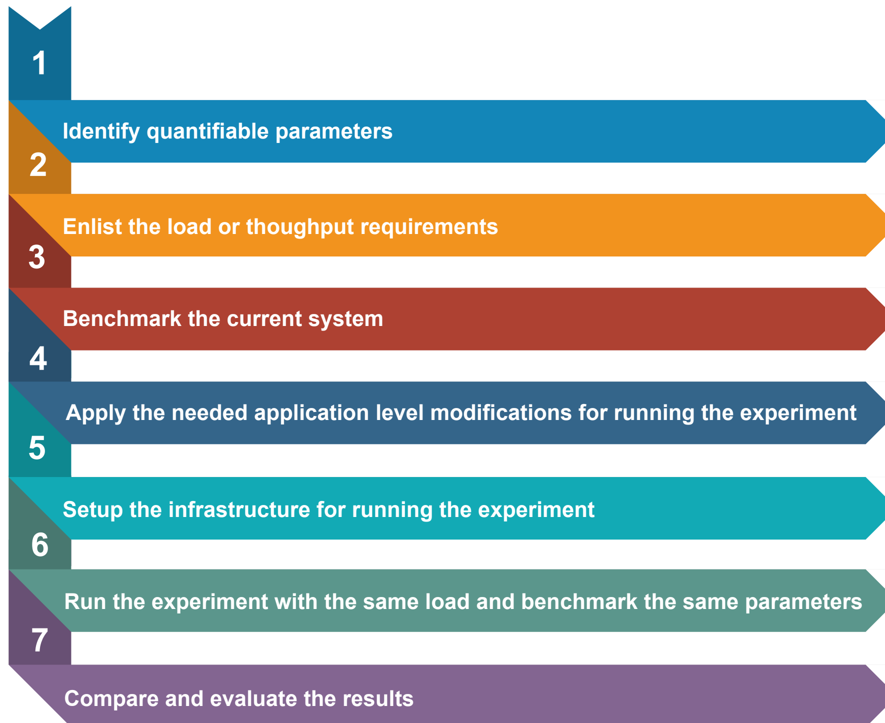


Figure 7: Experimentation Protocol

For the experiment to be conducted on one microservice, the whole research framework, with the first series of seven steps, needs to be applied once. The seven sub-steps of the sixth step in this framework would, however, have to be looped through for every feasible and possible enhancement solution. Let us look at these steps in more detail.

Understanding research framework (RF):

RF.1 *Gather Requirements:* List down the functional and the non-functional requirements of the microservice under investigation.

RF.2 *Evaluate in terms of CAP Theorem:* Evaluate all the requirements listed in step RF.1 in terms of the consistency, availability and partition tolerance. Using this evaluation, devise the CAP theorem requirements, for instance, does the microservice demand high availability, strong or weak consistency, etc.

RF.3 *Review existing literature in terms of Architectural Patterns:* Review the existing literature to identify architectural solutions and pattern implementations concerning the CAP Theorem restrictions listed in step RF.2.

RF.4 *Review existing literature in terms of databases:* Review the existing literature again to look for database level solutions. The questions aimed to be answered here are in the lines of which database to be considered, how to design it, with what configuration, etc., especially in accordance with the CAP Theorem restrictions listed in step RF.2.

RF.5 *List feasible solutions:* Converge the literature review performed in steps RF.3 and RF.4, and list down the feasible solutions for the microservice

under investigation. Important thing to note here is that feasibility is measured in terms of development and maintenance time, effort and cost, deployment cost, infrastructure cost, etc.

RF.6 *Setup and perform experiments:* For each of the solutions listed in step **RF.5**, design an experiment. An experiment is a prototypical implementation of the proposed solution, to test if it would actually perform as per theoretical predictions. More details about this step are explained in the experimental framework discussed later.

RF.7 *Evaluate results and make conclusions:* Evaluate the results obtained from each of the experiments performed in step **RF.6**, and compare them against each other. Further, use these evaluations to make a decision on the best available solution (to implement into the production system) under the given requirements and development conditions.

Here, **RF.6** demands the refactoring of the current system in accordance with the proposed solution, and its comparison against the current system. To explain the exact procedure for the same, we defined the Experimentation Protocol shown in Figure 7. As discussed above, the **RF.6** is split into subsequent steps, and the protocol is expected to be iterated upon as required. Understanding each of these subsequent steps of the experimentation protocol:

EP.1 *Identify quantifiable parameters:* List the parameters which can be used to compare the state of the system before and after the proposed changes. These can be response time, error rate, memory used, CPU load, etc.

EP.2 *Specify the load profile:* Explicitly state the foreseeable system requirements which are needed to run the system error-free with production load. These could be defined in terms of expected load, throughput, response time, data bandwidth, etc.

EP.3 *Benchmark the current system:* Run the current system, without any changes, with requirements specified in **EP.2**, and monitor the parameters specified in **EP.1**.

EP.4 *Apply the needed application level modifications for running the experiment:* Identify the application level changes needed for the proposed experiment, and perform them. Be aware to perform these changes in a separate copy (or branch) from the current deployed application code.

EP.5 *Setup the infrastructure for running the experiment:* Identify the infrastructure level changes needed for the proposed experiment, and perform them. Be aware to perform these changes in a separate deployment from the current deployment. It is also recommended to setup the entire experimentation setup in an isolated machine, so as to obtain standardise results unaffected by other entities in the system.

EP.6 *Run the experiment with the same load profile, and benchmark the same parameters:* Once the application and infrastructure changes are done, treat the deployment with the same amount of system requirements as decided in **EP.2**. Benchmark the parameters identified in **EP.1**, in the same form as was done for the current application in **EP.3**.

EP.7 *Compare and evaluate the results:* Compare the benchmarking results obtained in **EP.3** against the ones from **EP.6**. If the experiment promises improvements to the current system, consider it for critical analysis against all the experiments later.

Limiting the scope of this study, we will investigate only three microservices from the BranchKey architecture, namely, the Authenticator in Section 3.1, the File Uploader in Section 3.2, and the File Downloader in Section 3.3. In this chapter, we will apply the above discussed research framework and experimentation protocol on each of them. However, for better organisation, we will present the results of benchmarking the current system, and that of running every experiment in Chapter 5, and provide conclusions in Chapter 6. Simply put, for each iteration of experimentation protocol, **EP.3**, **EP.6** and **EP.7** is presented in Chapter 5, and for every application of the overall research framework, **RF.7** is presented in Chapter 6. Another thing to note is that **RF.1** demands the functional and non-functional requirements for the microservice under investigation. Since, we have already listed them down Chapter 3, we will not repeat them here. Summarising, **RF.1**, **RF.7**, **EP.3**, **EP.6** and **EP.7** are skipped from this chapter.

4.1 AUTHENTICATOR

As described in Figure 3, the current implementation of `authenticator` is a Python application with a PostgreSQL database. Let us look towards evaluating this microservice and proposing modifications for its enhancement.

Authenticator:RF.2: *Evaluate in terms of CAP Theorem*
Evaluating the functional and non-functional requirements for the authenticator, we converge over the following CAP requirements:

- Strong Consistency
- High Availability for Read APIs
- All data can be accommodated in a single partition

Authenticator:RF.3: *Review existing literature in terms of Architectural Patterns:*
`Authenticator` is a special example where it is acceptable to put all the data into a single partition. This means that `authenticator` as an individual microservice need not be treated as a distributed system. Thus, we can neglect the effects of CAP theorem herein, and focus on solutions which guarantee strong consistency and high availability within a single partition. We referred the literature for architectural patterns in microservices, to solve for a strongly consistent and highly available single node microservice.

[107] suggests three database patterns for a microservice. They are database per service, a database cluster and a shared database server. A database cluster is basically a separate cluster of database nodes, which store data for all the microservices in the system. Although the data storage is shared, data is organised in multiple databases, each privately accessible only to the owner microservice. It is the same as a shared database server, but in a distributed format. It promises more availability and reliability compared to a single database server, but keeping all the data in one place could again lead to

a critical data bottleneck in the entire system. Also, it forces all data intensive services to use one database management system. This is clearly not the case in BranchKey. Here, every service uses a different database management system. This makes the creation and maintenance of such a shared database server/cluster pointless. As far as the third pattern, i.e., database per service is concerned, `authenticator` already implements that.

Another highlighted pattern in the literature is DBaaS, i.e., database as a service pattern. As described in [77] and [76], it creates the database into an independent microservice of its own. Implementation wise, this means that we use the database service provided by the cloud platform provider. In BranchKey's case, this gives the ownership of the database management server to the external cloud provider AWS. This also demands that business logic be implemented within the database layer, and eliminates the need for corresponding application domain layer. Implementing business logic into databases is very complex to do, and maintain. Also, this transfers the control of the database to the cloud platform, thus making the possibility of migration to another cloud platform intensely complicated. Thus, this approach is also discarded.

As a fix to the issues of this pattern, [121] proposes a *Trinity Model*. This involves a load balancer, a sharded relational database management system, a replication server and a non-relational NoSQL distributed database. The idea behind this configuration is that all data is stored in the sharded relational database in a structured format, and is asynchronously replicated to the non-relational database by the replication server. It promises a high availability and scalability, along with the ACID properties of a relational database. It hopes to serve all the online transaction processing queries from the relational database, and all the online analytical processing queries from the non-relational database. As promising as this model sounds, it is an over-engineered solution for the problem at hand. Firstly, given the small size of data, we do not need a distributed system for this use case. Secondly, analytical processing on this data is not specified as a requirement. Thirdly, sharding of the relational database can potentially reduce the consistency of the data. Thus, this approach is rejected.

Authenticator:RF.4: *Review existing literature in terms of databases:*
The user data and the session management data to be stored in `authenticator` is an explicit example of tabular format, where tables would be connected via foreign keys, and primary keys and complex indexes would be needed to perform insert, update and lookup queries. Such a configuration is not natively supported by NoSQL databases. Thus, we will narrow down our search for relational databases. [50] lists 24 design patterns for relational databases. Some of them are of importance for our use case, like the "Created When" and "Updated When" to log and analyse data accesses. The "Record Status" pattern can be used for storing the status of a user's login activity. The "Session" pattern can be used to log when exactly a record was created and updated. However, these are all already implemented within the data model for the `authenticator`.

[57] gives a detailed analysis of 80 databases. Filtering the search to relational databases which promise consistency and availability, and are freely available, we narrow down to PostgreSQL as the only recommended option. Thus, we select PostgreSQL for our use case, which is also already a

part of the current implementation. [57] also suggests that in a relational database, we scale read operations using a master/slave architecture. Currently, `authenticator` does not follow this, making this a valid approach for an experiment.

[102] suggests client side caching as a solution for faster fetch data queries. In BranchKey's case, the clients for `authenticator`'s data are either the external users, or the `api-gateway`. If we maintain client side cache for read queries, it has to be at `api-gateway`. According to the flow in Figure 2, every time a user logs in, the `authenticator` somehow needs to communicate the new access token to the `api-gateway`. This falls outside the responsibilities of `authenticator` and violates the Single Responsibility Principle [75] for microservices. Also, this makes the `api-gateway` stateful, which is an anti-pattern. Thus, we reject the idea of client side caching. However, this opens up a discussion for server side caching.

Authenticator:RF.5: *List feasible solutions:*

We can narrow down the research from **Authenticator:RF.3** and **Authenticator:RF.4** to the following two feasible solutions:

- The first alternate solution for `authenticator` can be designed to provision a master/slave architecture for its database. This can be exploited to serve read queries faster by pointing the read queries to slaves only, while only the write queries go to the master database instance. This also makes the two independently scalable, as we can provision many slaves for a single master.
- The second solution can explore server side caching for storing the data for read queries. We can maintain an in-memory cache at the `authenticator` to serve the read queries faster. This data would be local to the `authenticator`, and thus can be updated with every write query. Although, whether this will actually reduce the response latency, and be cost efficient, needs to be experimented through.

Authenticator:RF.6: *Setup and perform experiments:*
We perform two experiments:

- Master/slave configuration for the database, with write queries directed to master and read queries directed to slave
- Server side caching

An iteration of experimentation protocol is performed for each of these, as described below.

4.1.1 *Experiment 1: Master/slave DB distribution*

Let us now apply the experimentation protocol described in Figure 7 to the first experiment for `authenticator`.

Authenticator.1:EP.1: *Identify quantifiable parameter:*
The primary aim of investigating the `authenticator` is to reduce the response time and the error rate for the desired throughput. Thus, these are the most important metrics to be measured in this experiment. We observe *API response time* to identify a total of the processing lag introduced by the application logic and the time taken by the database to serve the corresponding data query. This would help us compare different database level strategies, along with their impact on the application logic layer. Since availability is one of the CAP requirements for the `authenticator`, we will monitor the *error rate* for the API calls. Increased internal server errors would directly translate to low availability here.

In addition, we also monitor the *CPU load* and the *memory usage* of the database instance and the service instance. This would help us identify the optimal horizontal scaling configuration.

Authenticator.1:EP.2: *Specify the load profile:*
As described in Section 3.1, we have two throughput requirements for `authenticator`:

- Write queries with at least *25 requests per second*, with a response time of at most *200 ms*
- Read queries with at least *25 requests per second* with a response time of at most *200 ms*

Important thing to note here is that the *write* queries refer to the *register, login* and *logout* APIs, while the *read* queries refer to the *authenticate* API.

Authenticator.1:EP.4: *Apply the needed application level modifications needed to run the experiment:*
There is no application level modification needed for this experiment.

Authenticator.1:EP.5: *Setup the infrastructure for running the experiment:*
Following infrastructure level modifications are performed for this experiment:

- We first create a master/slave configuration of the PostgreSQL database. This is implemented using Kubegres [64].
- We need to separate the database configuration of the service, so that it calls master database for write queries, and slave for reads. To avoid propagating this into the application logic, we rather deploy two different instances of the `authenticator` service, one of which points to master database and other points to the slave.
- Since BranchKey uses Kubernetes, we deploy these two service instances as two different deployment objects. The clients for register, login and logout requests call the former instance, while the clients for authenticate requests call the latter instance. This separation comes easy in the current architecture because a client does not make both calls. Write calls come only from external users, while read calls come only from the `api-gateway`.

4.1.2 *Experiment 2: Server side caching*

Let us now apply the experimentation protocol described in Figure 7 to the second experiment for authenticator.

Authenticator.2:EP.1: *Identify quantifiable parameter:*
Same as **Authenticator.1:EP.1**

Authenticator.2:EP.2: *Specify the load profile:*
Same as **Authenticator.2:EP.1**

Authenticator.2:EP.4: *Apply the needed application level modifications needed to run the experiment:*
We need the following application level modifications for this experiment:

- Every time the login API is called, a database write for *login-details* and *user access-token* happens. To employ server side caching, we create a cache entry for this data if it does not exist, or update it if it already exists.
- Every time the authenticate API is called, the service first checks the cache. If the cache contains the relevant entry, it is used to serve the client request. This saves a read call to the database. However, if for some reason there is no cache entry, the database is called, and a cache entry is created in accordance with the data fetched.

Authenticator.2:EP.5: *Setup the infrastructure for running the experiment:*
The following infrastructure setup is needed to perform this experiment:

- We need to setup an in-memory caching database. We use Redis for this.

One thing to note here is that we limit the scope of this experiment by only working with a vanilla Redis configuration. However, it can be explored further by implementing a master/slave in Redis, or a Redis cluster, or even Redis Sentinel configuration if needed. Moreover, other in-memory databases apart from Redis can also be considered.

4.2 FILE UPLOADER

As described in Figure 4, this microservice is currently implemented as a GoLang application. It uses Redis as caching database, and RabbitMQ as a message broker for queuing file upload events and facilitating asynchronous communication with the downstream services. Let us look towards evaluating this microservice and proposing modifications for its enhancement.

File Uploader:RF.2: *Evaluate in terms of CAP Theorem*
Evaluating the functional and non-functional requirements for the file uploader we converge over the following CAP requirements:

- Eventual Consistency
- High Availability
- Partition tolerance or bounded response latency

File Uploader:RF.3: *Review existing literature in terms of Architectural Patterns:* This microservice needs to be designed in compliance with high availability and partition tolerance requirements. This means that, unlike `authenticator`, the `file-uploader` service instance itself can be deployed across different networks (or *availability zones* in cloud terminology), or even on the very lowest level, the database can be distributed over many nodes. According to the current BranchKey architecture in Figure 2, this service not only needs to be highly available for client communications, but to also facilitate easy data access for the downstream consumers of the input file data. [99] and [74] suggest asynchronous inter process communication for better data availability to downstream services. This does theoretically validate the current data flow, which implements a *server-worker* pattern to consume data from `file-uploader` in an asynchronous manner, and send it to the `central-aggregator` asynchronously as well. The authors of [74] also suggest *service registry* pattern for highly available communication amongst different services in a network. Since BranchKey uses Kubernetes as a deployment tool, *service registry* comes by default therein.

[113] proposes that providing a read-only data view to the data accessing services reduces the need for data replication, pub-sub communication, etc. The current implementation already follows this where `central-aggregator` reads the file data from the `file-uploader` Redis directly. It is not published to any secondary place. This risks the security of the data and violates the data ownership rules for microservices. Thus, we should configure a replica or a slave of the master Redis database, and let aggregator access this slave only. [43] also suggests using master/slave replicas. [43] says that although replication introduces temporary inconsistencies, but it also allows for better availability and performance. These inconsistencies are very short lived in cases of live replication, and are thus eventually corrected. As long as eventual consistency with a bounded data synchronisation delay is acceptable, replication should be implemented.

The current vanilla Redis implementation is a single node instance which has no persistence. This is a potentially critical bottleneck, which in case of failure, can not only cause system downtime, but also lead to loss of all data. Thus, another promising strategy to explore in this regard would be a Redis cluster, along with background data persistence. Data persistence, however, can lead to slowed down response times from Redis [91]. Hence, we can design an experiment to test whether redis-cluster with persistence is a feasible solution or not.

File Uploader:RF.4: *Review existing literature in terms of databases:*
The data in `file-uploader` is not of the tabular form. It is a use case for the caching of input files from users, and some of the index data for each of these files. Neither the file, nor the index data is structured. Thus, we are going to narrow down our search for solutions into the domain of non-relational or NoSQL databases. Focusing on NoSQL databases compatible with the CAP requirements of file uploader, we research the literature for provisions with high availability and eventual consistency. An important thing to note here is that although strong consistency is not needed, eventual consistency is expected. The system cannot afford to lose data. Keeping these in mind, some solutions propose ways to implement eventual consistency at down at the database design level. However, they are discarded for this project because devising changes at that level and ending up designing

an entirely new database management system is beyond the scope of this project. This leaves us to focus on solutions which promise high availability instead.

[94] gives an example of the practical industrial use of a NoSQL databases to ensure availability by relaxing consistency guarantees to eventual consistency. They use Cassandra database with three replicas. However, for our use case, Cassandra does not fit because we do not have columnar data. As discussed above, file uploader data is a key-value kind. Thus, the current implementation already uses Redis, which is also a NoSQL database.

[10] explains in detail how Redis is in fact better suited for our use case as compared to other key-value in-memory cache databases like Memcached and Aerospike. The most distinguishing property of Redis over other data stores considered here is largest key and value size allowed (512 MB). Although Memcached provides multi-threading client handling, and lower latencies in read-heavy applications, and Aerospike provides lowest memory footprint amongst the three, Redis is the best option for complex value types like the file data storage in `file-uploader`.

Further, for the asynchronous message passing, [33] suggests using a RabbitMQ cluster with *at least once* delivery guarantee. Although the current system still uses a RabbitMQ, it is a single instance deployment. We can experiment with a cluster to achieve higher availability and increased throughput. The researchers of [33] say that *"should a RabbitMQ node terminate or become unavailable due to a network partition, the cluster will automatically handle partitioning based on its consistency configuration"*. They also claim that such a scaled asynchronous communication can ease horizontal scaling on the consumer end of the queue. In BranchKey's pipeline, multiple `aggregation-task-creators` and `central-aggregators` can be deployed to enhance concurrent processing. Overall, [33] comments that such horizontal scaling, data replication, and clustering improves availability of the system. They have also used Redis cluster as one of the components.

File Uploader:RF.5: *List feasible solutions:*

We can narrow down the research from **File Uploader:RF.3** and **File Uploader:RF.4** to the following three feasible solutions:

- The first solution would compare the current single instance Redis with a master/slave Redis configuration, and background data persistence. Important point of observation here would be if slave replication or background data persistence adds any delay into the Redis response time. All other benefits of a slave, such as data security and better data ownership are already provided.
- The second experiment would test the feasibility of Redis cluster along with slave configuration and background data persistence. Again, we examine if this introduces any delay in the Redis response time. Another highly available Redis configuration is Redis Sentinel Cluster [90]. It deploys sentinel nodes to auto-promote one of the slaves to master, in case the previous master node fails. However, this does not allow for multi-master configuration. Thus, there is no point testing this if our first experiment of single master-slave setup does not prove

to be viable. Hence, we scope the Redis Sentinel experiment into future work, under the condition that first experiment proves to be promising for the `file-uploader` requirements.

- We could also evaluate RabbitMQ cluster, or even compare RabbitMQ to another event queue like Kafka.

File Uploader:RF.6: *Setup and perform experiments:*
We perform two experiments:

- Master/slave configuration for Redis with data persistence
- Redis cluster with replication and data persistence

We do not perform experiments to evaluate the third solution because changes from this would mostly be experienced on the consumer end of these message queues. That would mean changing and testing the whole BranchKey system at once, which is beyond the scope for this project. We are only focusing on solutions which can be tested on one individual microservice.

4.2.1 *Experiment 1: Redis Master/Slave*

We will now apply the experimentation protocol described in Figure 7 to the first experiment of the `file-uploader` evaluation.

FileUploader.1:EP.1: *Identify quantifiable parameters:*
The most important parameter to be observed here is the *response-time* from the `redis-cache` (in Figure 4) as well the overall *response-time* of the service for upload file API calls. This allows us to quantify the latency of the service. This experiment will however also include the delay for replication and the persistence, and put us in a position to evaluate if the *master/slave* configuration leads to any extra delay in the API response, and if that delay is acceptable or not.

Secondly, we plan to monitor the *error-rate* on the upload file API calls, and segregate these into service errors and `redis` errors. The *error-rate* parameter directly translates to the availability of the service as well as that of the `redis`. In case `redis` becomes unavailable, how much time does it take for promoting the slave to master, and how difficult is that, would be some questions we can answer with this.

Thirdly, we can monitor the *CPU load*, the *memory load*, and the *memory fragmentation* of the `redis` instances (both master and slave), to keep an eye on the requirements for scaling. With this configuration, horizontal scaling is not possible. If vertical is scaling needed, and if it is feasible or not would be answered here.

FileUploader.1:EP.2: *Specify the load profile:*
As discussed in Section 3.2, we have two throughput requirements for `file-uploader`:

- File upload queries with at least *20 requests per second*, with a response time of at most *400 ms*

- File size varying up to *1 MB*

FileUploader.1:EP.4: *Apply the needed application level modifications needed to run the experiment:*
There is no application level modification needed for this experiment.

FileUploader.1:EP.5: *Setup the infrastructure for running the experiment:*
We need to provision a slave for the existing Redis master. Additionally, we configure data persistence. As explained in the official Redis documentation [89], there are multiple ways to configure data persistence. We use the combination of *Redis Database (RDB)* and *Append Only File (AOF)*, to get the best of both worlds.

4.2.2 *Experiment 2: Redis Cluster*

We will now apply the experimentation protocol described in Figure 7 to the second experiment of the `file-uploader` evaluation.

FileUploader.2:EP.1: *Identify quantifiable parameters:*
The prime strength of Redis cluster over single node Redis is the ability to scale horizontally. Thus, in addition to all the parameters mentioned in **FileUploader.1:EP.1**, we monitor the system health metrics for all the Redis cluster nodes as well. This will help us understand the needs for horizontal scalability in the cluster.

FileUploader.2:EP.2: *Specify the load profile:*
Same as in **FileUploader.1:EP.2**.

FileUploader.2:EP.4: *Apply the needed application level modifications needed to run the experiment:*
The only application level change needed here is to switch the Redis client from single node to cluster client.

FileUploader.2:EP.5: *Setup the infrastructure for running the experiment:*
We need to perform the following actions in the infrastructure layer:

- Setup the Redis cluster, with at least 3 nodes.
- Configure a slave for each of the nodes in the cluster.
- Configure background data persistence for all the nodes, with the *RDB + AOF* configuration as explained in **FileUploader.1:EP.5**.

4.3 FILE DOWNLOADER

`File-downloader`, as shown in Figure 5, is currently a GoLang application with a PostgreSQL database. Current implementation stores file indexing data in PostgreSQL, and the actual files in the local file system. It also uses RabbitMQ for the purposes of asynchronous communications as needed. Let us look towards evaluating this microservice and proposing modifications for its enhancement.

File Downloader:RF.2: *Evaluate in terms of CAP Theorem*
Evaluating the functional and non-functional requirements for the authenticator, we converge over the following CAP requirements:

- Strong Consistency
- Eventual/Low Availability
- Partition tolerance or bounded response latency

File Downloader:RF.3: *Review existing literature in terms of Architectural Patterns:*
In the spectrum defined by CAP theorem for distributed systems, `file-downloader` prefers strong consistency over high availability. This is of course in addition to partition tolerance. Reduced availability implicitly reduces the performance of the system, for if network partitions, fulfilment of consistency guarantees can take infinite latencies. Even [54] says that in a physically distributed cloud application, strongly consistent systems are not the most available or scalable ones. In fact, many data storage formats don't even support strong consistency for cloud based applications distributed across data centers. [54] mostly promotes eventual consistency, especially in cases of replicated data sets. In fact, it suggests to split data according to the requirement identification, such that each split can implement a different policy with regards to data consistency. This allows to implement strong consistency only when absolutely needed. In our case, we can have it only for the file indexing data stored in PostgreSQL database, and not for the actual files stored in the file system. This reduces down the scope for maintaining strong consistency.

Another thing suggested in [54] is that a strongly consistent system can afford to be eventually consistent, with some possible inconsistencies while processing the transactions. In `file-downloader`, the requirement states that the system should be consistent for the user to be able to download a file. Before that it can, however, afford eventual consistency. Thus, we can perform the intermediate transactions of creating a record entry in the PostgreSQL database and storing the file in the file system separately. This allows for eventual consistency in the system before finally sending out an event to the end user to avail the file for download. This also opens up opportunities to handle the failed transactions in an idempotent manner before a user hits the download API. Once data is consistently stored across all nodes, users can access it with strong consistency guarantees. This architectural solution is also already implemented in the current system.

File Downloader:RF.4: *Review existing literature in terms of databases:*
As described in **File Downloader:RF.3**, we can make the download API strongly consistent even with eventual consistency within intermediate transactions in the `file-downloader`. This means that two steps of storing the file records in PostgreSQL database, and storing the file in the file system can be individually eventually consistent. Referring back to the literature for eventual consistency models in relational databases, [100] proposes many data replication schemes for cloud applications. However, all of these schemes suggest changes on the base database implementation level, which is beyond the scope of this project. Thus, we restrict ourselves to the inbuilt

replication strategy of the PostgreSQL database. The current single node implementation of PostgreSQL database promises strongly consistent ACID transactions by default. We can however, evaluate if replication adds any value to the system by comparing its performance with the current single node implementation.

[62] suggests a *Scalable Distributed Two-layered Data Structures (SD2DS)* architecture for achieving strong consistency with NoSQL databases like Mongo or Memcached. Similarly, [119] suggests a 2-phase commit protocol to provide ACID transactions with distributed NoSQL databases. As promising as these solutions seem, their infrastructure requirements are far greater than the current scalability requirements of the BranchKey system. Thus, these vastly complex implementations are bound to be an example of over-engineering in this context. Hence, we discard these approaches, and limit ourselves to the replication strategies provided by PostgreSQL database for now.

As far as alternatives for PostgreSQL are considered, [88] suggests some databases for consistent and partition tolerant database management systems which are freely available. However, all of them are either key value stores, or document oriented stores, or wide column stores. None of these fit the tabular data structure of a file record to be stored in file downloader. Thus we stick to PostgreSQL. Although the native deployment of PostgreSQL is not partition tolerant, we can play around by provisioning some replication slaves. Moreover, this data need not be stored across partitions anyway. Files would have to be stored across distributed nodes.

PostgreSQL also supports table partitioning [106]. As [3] suggests, horizontal table partitioning can be used to tackle big datasets in relational databases with optimal efficiency. In the case of `file-downloader`, we can partition the file indexing data, and look for performance enhancements. Since all the partitions are stored in the same node, it does not compromise the consistency of the system, even in case of network partitions. On the contrary, since each partition is now smaller than the entire table, it promises faster response times from the database.

File Downloader:RF.5: *List feasible solutions:*

We can narrow down the research from **File Downloader:RF.3** and **File Downloader:RF.4** to the following three feasible solutions:

- As a first solution, we can explore the effects of adding a replica to the existing PostgreSQL database in the current system. Using the native master/slave provision in PostgreSQL, we can direct write queries to the master database and the read queries to the slave database.
- Secondly, we can examine the pragmatic enhancements offered by table partitioning in PostgreSQL database. We can horizontally partition the data using the hash partitioning technique for the group-id field. This not only promises to speed up the get data queries, but also provisions for better data management and archival.
- Lastly, we can evaluate different file storage systems and perform a comparative study as in terms of their fault tolerance, response latency and data consistencies.

File Downloader:RF.6: *Setup and perform experiments:*

We perform two experiments:

- Replication on PostgreSQL using master/slave configuration, with write queries directed to master and read queries directed to slave only
- Horizontal table partitioning by hashing the group-id field

We do not perform the experiment to evaluate multiple file storage systems because the changes needed for that are beyond the scope of this project. Additionally, we could evaluate cloud storage systems, but BranchKey has already discarded the use of AWS S3 as an external file storage. This is in order to avoid vendor lock in. As far as other distributed file systems like Mongo GridFS or IPFS is concerned, they are relatively complicated to integrate and evaluate for this project.

4.3.1 *Experiment 1: Data Replication*

Let us now apply the experimentation protocol described in Figure 7 to the first experiment for `file-downloader`.

FileDownloader.1:EP.1: *Identify quantifiable parameters:*

Strong consistency is the most important requirement of the `file-downloader`. So, we plan to monitor the *error rate* on the API calls. We can segregate these errors into errors of data consistency and errors of service unavailability. Lower the number of corresponding errors would mean stronger data consistency on the backend.

The second most important parameter is the response latency, which will be measured in terms of overall *response-times* of the APIs and the individual response times from the file indexing and the file storage databases. This is where we notice the performance enhancement provided by the PostgreSQL configuration under evaluation, if any.

Thirdly, we can monitor the system health metrics on the database instances, mainly the *CPU load*, and the *memory usage*. If only read queries are adding to load, we have an option to horizontally scale the number of slaves and deploy service instances to only answer read data queries using those slaves. However, if write queries are the main ones contributing to the system load, we might have to explore vertical scaling. Horizontal scaling is not an option in that case because it is a single master database.

FileDownloader.1:EP.2: *Specify the load profile:*

Based on the requirements stated in Section 3.3, we identify three throughput requirements for `file-downloader`:

- Aggregated file save queries with at least *5 requests per second*, with a response time of at most *50 ms*
- File download queries with at least *5 requests per second*, with a response time of at most *10 ms*
- File size up to *1 MB*

FileDownloader.1:EP.4: *Apply the needed application level modifications needed to run the experiment:*

For this experiment, we just need to change the service layer to accept both, the master and the slave database instances, and redirect write and read data queries accordingly.

FileDownloader.1:EP.5: *Setup the infrastructure for running the experiment:*

On the infrastructure level, we need to provision a master/slave configuration for the PostgreSQL database.

4.3.2 *Experiment 2: Data partitioning*

Let us now apply the experimentation protocol described in Figure 7 to the second experiment for `file-downloader`.

FileDownloader.2:EP.1: *Identify quantifiable parameters:*
Same as **FileDownloader.1:EP.1**

FileDownloader.2:EP.2: *Specify the load profile:*
Same as **FileDownloader.1:EP.2**

FileDownloader.2:EP.4: *Apply the needed application level modifications needed to run the experiment:*

We need to perform the following application level modifications:

- Firstly, we need to change the database schema to create hash partitions for all the group-ids. Since group-ids are UUIDs, we will hash them into 16 buckets for now. If this approach proves to be viable, we can change this number to a larger value.
- Secondly, we need to change all the database queries to also incorporate group-id, to help redirect to the correct table partition.

FileDownloader.2:EP.5: *Setup the infrastructure for running the experiment:*
We do not need any infrastructural changes for this experiment.

5 EVALUATION

In this chapter we will discuss the results from each of the experiments described in Chapter 4, and evaluate them against the set criteria.

To subject each of the systems with desired load requirements, we created a `load-test` script. It is a GoLang application which hits the API to be tested with configured requests per second rate. This `load-test` script exposes APIs to start and stop the load tests for every microservice. Each load test launches multiple *goroutines* [109]. Each of these *goroutines* is a REST client which fires an HTTP request to the microservice under investigation, and records the response status. This `load-test` application is also deployed as a Kubernetes service in the same cluster to avoid additional network delays. However, for the sake of isolating this system and keeping it from interfering with the actual BranchKey platform, it is deployed on an separate node. This is done via node tainting and toleration [108].

To evaluate the performance parameters of the system, we deploy a `monitoring-stack` within the BranchKey ecosystem, as shown in Figure 8. It consists of *Prometheus* [87] and *Grafana* [47]. Each of the microservices exposes business metrics onto a REST endpoint, which is scraped by *prometheus* metrics scrapers at regular intervals. This data is then visualised into graphical format on *grafana* dashboard.

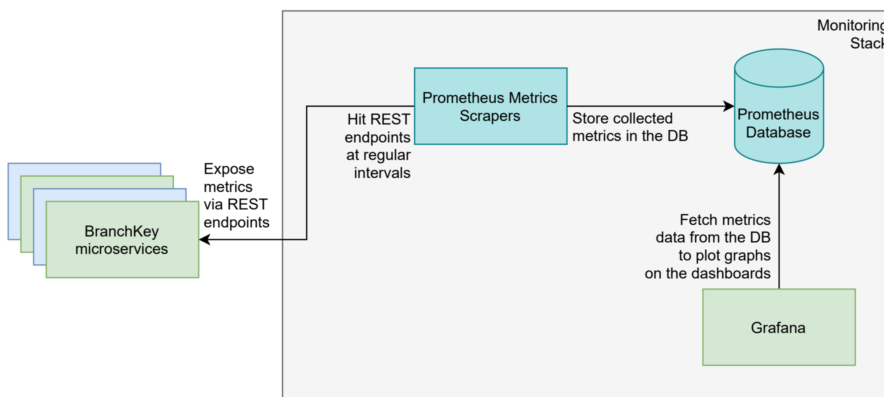


Figure 8: Monitoring Stack

We also evaluate the system in terms of its infrastructural cost. Currently, BranchKey uses Amazon Web Services cloud platform to deploy the system. Table 1 gives the pricing model and an estimate for the monthly cost for the same.

This infrastructure cost highly depends on two variables: the computation power and the memory usage of the system. Table 2 evaluates the current technological stack in terms of its bare minimum memory footprint in no-load state and no-data state. Quantitatively, this is the minimum memory needed to horizontally scale an instance of the corresponding type. A key observation made here is that a Golang application is a about 9 times more

Entity	Rate	Monthly Cost[1]	Source
Amazon EKS cluster	`$ 0.10` per hour	`$ 73`	[7]
Classic Load Balancer	`$ 0.028` per hour + `$ 0.008` per GB of data processed	`$ 27.45`[2]	[38]
Amazon EC2 On-Demand[3]	`$ 0.0368` for EC2 + `$ 0.11` per GB for ESB	`$ 116.26`[4]	[82]
Total Cost		`$ 216.71`	[1]

1 Considering `730` hours per month
2 For `10` agents uploading and downloading a file of `1 MB` each, performing `1` aggregation per minute = `10 * 2 MB = 20 MB` per minute
3 `t3.medium: 2 vCPUs, 4GB` Memory with an Elastic Block Storage Disk: `20 GB gp2`
4 For `4` instances

Table 1: AWS Pricing Model

memory efficient than a Python application. Hence, the redevelopment of the `authenticator` into a Golang application can be a future consideration.

Entity	Memory Footprint
Typical Python Application	`44 MB`
Typical Golang Application	`5 MB`
PostgreSQL DB	`34 MB`
Redis	`7 MB`
RabbitMQ	`110 MB`

Table 2: Memory Footprint of the Current System

5.1 AUTHENTICATOR

We subjected `authenticator` to throughput of about *45 requests per second* with the following load on each of the following APIs:

- Write APIs: 20 requests per second
  - Login API: 15 requests per second
  - Logout API: 1 request per second
  - Signup API: 5 requests per second
- Read API: 25 requests per second
  - Authenticate API: 25 requests per second

The experiment was run for a total duration of `30` minutes. The system was seeded with the credentials of 25 users to begin with. Following subsections present a detailed evaluation of the system performance under these conditions, with each of the prescribed architectural solutions. For the sake of these experiments, we assumed that the rest of the BranchKey platform performs ideally. Thus, the downstream dependency over the `authoriser` and the upstream dependency over `api-gateway` do not affect the performance of this system.

5.1.1 *Current System*

The current system uses a single instance of service pod deployment to serve all the requests. An overview of its performance parameters is shown in Figure 9. This service pod is backed by a single instance PostgreSQL deployment, which performed as per Figure 10 and Figure 11. The graphs develop in spikes because of small sleep windows triggered in the load tester in between the requests, so as to control the number of open and connections and the system resources in use.

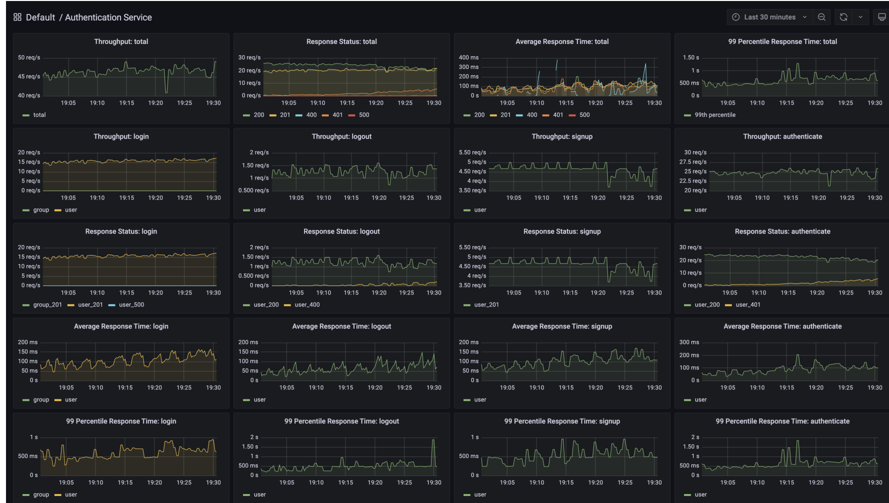


Figure 9: Authenticator Dashboard [Current System]

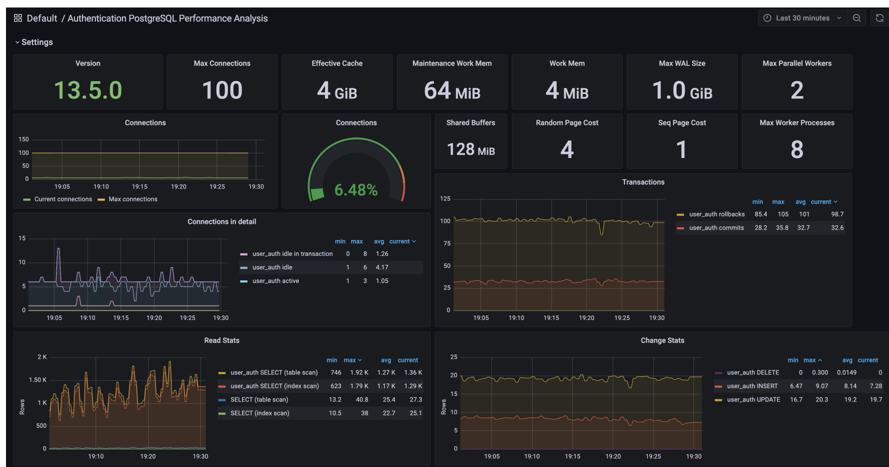


Figure 10: Authenticator PostgreSQL Dashboard [Current System]

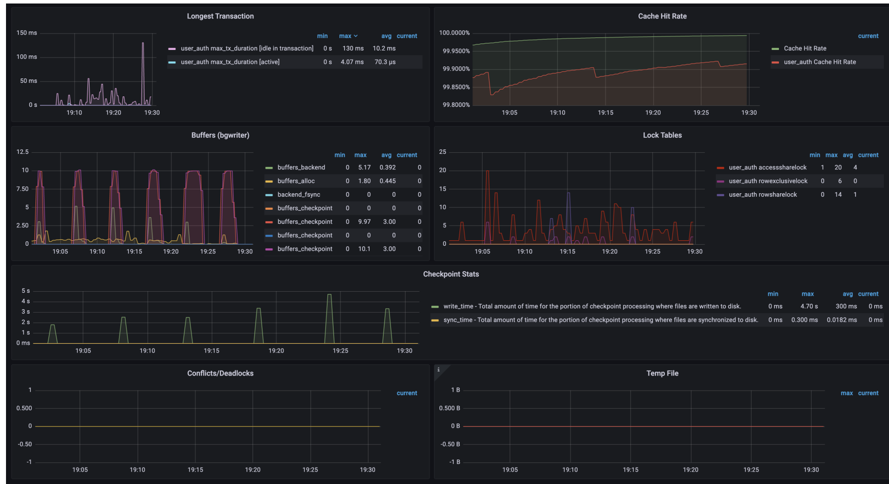


Figure 11: Authenticator PostgreSQL Dashboard Continued [Current System]

Analysing the *response-status* for each of the APIs in Figure 9, we see that the error rate from all the APIs is absolutely *zero* for server errors. This means that the current implementation provides high availability. Also the *response-times* are bounded within `150 ms` for both read and write APIs.

Evaluating in terms of system health metrics, we see that the CPU load on the service pod is less than `0.7` as shown in Figure 12, and that for database pod is less than `0.1` as shown in Figure 13. In terms of process memory usage, the service uses about `325 MB` as presented in Figure 14 where as the database uses only about `53 MB`, as shown in Figure 15. Also the persistent volume usage of the database is only `130 MB` as shown in Figure 16. Overall, the system is healthy, and beyond the need for horizontal scaling currently.

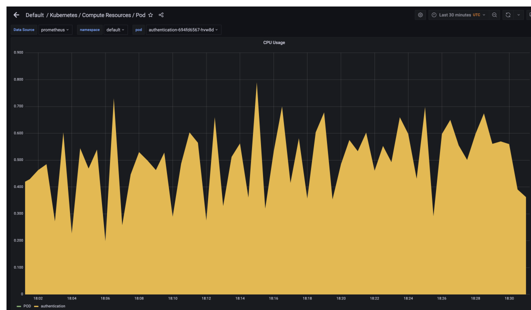


Figure 12: Authenticator Service Pod CPU Load [Current System]

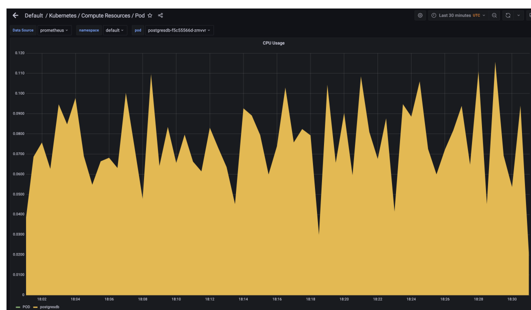


Figure 13: Authenticator PostgreSQL Pod CPU Load [Current System]

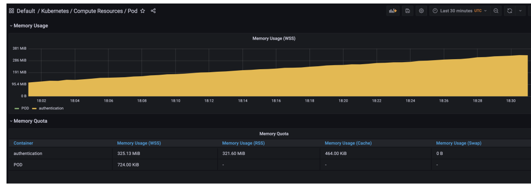


Figure 14: Authenticator Service Pod Memory Usage [Current System]

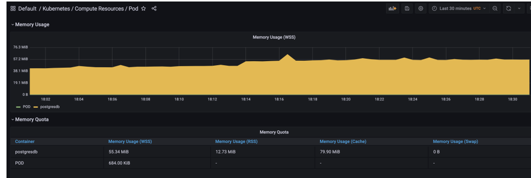


Figure 15: Authenticator PostgreSQL Pod Memory Usage [Current System]

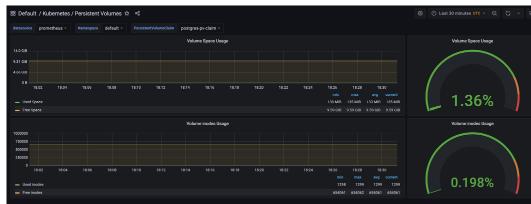


Figure 16: Authenticator PostgreSQL Persistent Volume Usage [Current System]

5.1.2 *Results from the experiments*

Let us now enlist the results obtained from the experiments performed over the `authenticator`.

5.1.2.1 *Experiment 1*

The first experiment evaluated the efficiency of a *master/slave* architecture for the PostgreSQL database. When subjected to the defined load requirements, service performed as shown in Figure 17. Evaluating the quantifiable parameters, we can see that service returns absolutely *zero* unavailability errors for all the APIs. In fact, secluding the database write queries to *master* and the read queries to the *slave* instance has resulted in a vast improvement in the *response-time* of all the APIs. All the write APIs now respond within `100 ms` where as the read APIs take only `35 ms`.

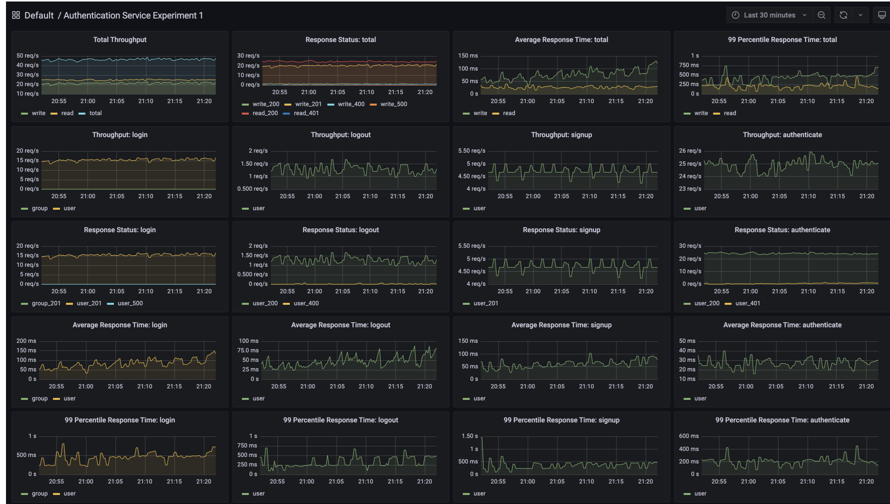


Figure 17: Authenticator Dashboard [Experiment 1]

Further, we evaluate the service and the database instances in terms of their system health metrics, i.e., *CPU Load* and *memory usage*. For the service pods, the *CPU Load* of both the service instances, shown in Figure 18 and Figure 19, is less than that of single service instance in the current system. Summed together, they both have a *CPU Load* of `0.650`, which is `7.14%` less than the current system. Similar trend is followed for the two database instances as well, as plotted in Figure 20 and Figure 21. Even with the additional load of maintaining data replication, each of these instances uses less computation power. Numerically speaking, the *master* has a *CPU Load* of `0.02`, and the *slave* has a `CPU Load` of 0.025. Together, they are still `55%` less than the database of the current system. This can be highly attributed to the distribution of queries across the instances. Discussing the memory usage for the service shown in Figure 22 and Figure 23, although the individual memory requirements of the two service deployments are each less than the single service deployment of the current system, their combined value in fact adds up to slightly bigger value of `330 MB`. For the database deployments, although the slave instance is low on memory usage (Figure 25), the master instance takes up `13 MB` more memory instead (Figure 24). As far as persistent volume is considered, the databases, as shown in Figure 26 and Figure 27, individually use less memory than the current system, but totally add up to `220 MB`, which is about `62%` more than the persistent volume usage of the current system. However, this can be optimised further by removing a persistent volume storage from the *slave*. This is because *slave* is replicating the same data as the *master*. We do not need another persistent storage of the same data.

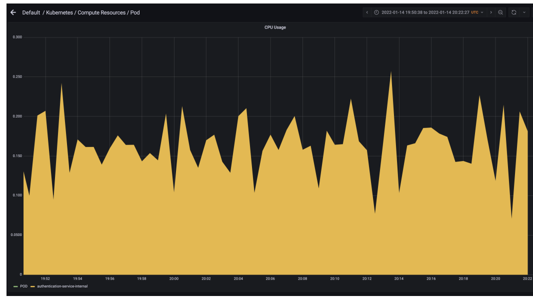


Figure 18: Authenticator Read Service Pod CPU Load [Experiment 1]

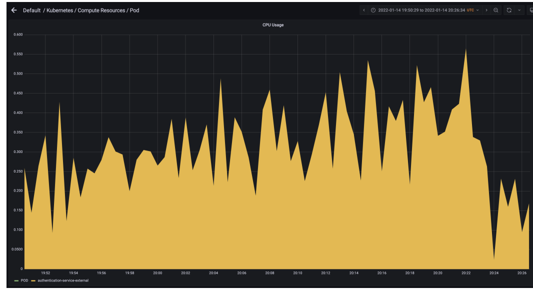


Figure 19: Authenticator Write Service Pod CPU Load [Experiment 1]

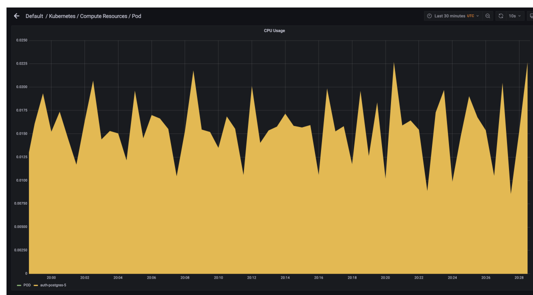


Figure 20: Authenticator Master DB Pod CPU Load [Experiment 1]

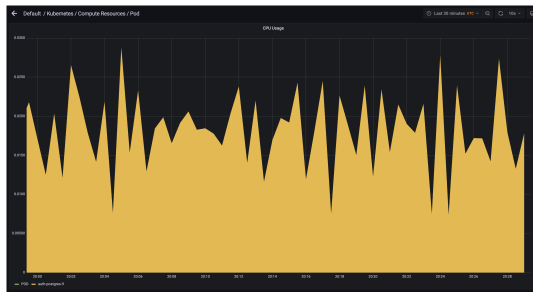


Figure 21: Authenticator Slave DB Pod CPU Load [Experiment 1]

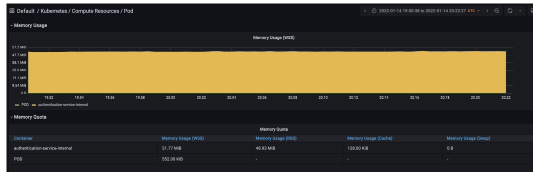


Figure 22: Authenticator Read Service Pod Memory Usage [Experiment 1]

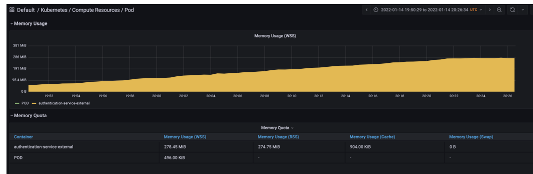


Figure 23: Authenticator Write Service Pod Memory Usage [Experiment 1]

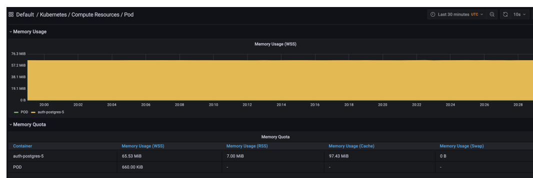


Figure 24: Authenticator Master DB Pod Memory Usage [Experiment 1]

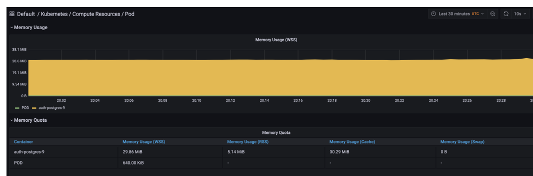


Figure 25: Authenticator Slave DB Pod Memory Usage [Experiment 1]

Figure 26: Authenticator Master DB Persistent Volume Usage [Experiment 1]

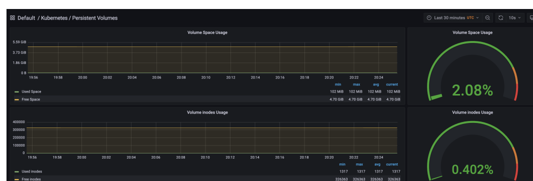


Figure 27: Authenticator Slave DB Persistent Volume Usage [Experiment 1]

5.1.2.2 *Experiment 2*

This experiment explored the efficiency and feasibility of server-side caching for serving the read queries in the `authenticator`. In addition to monitoring the database in a dashboard similar to Figure 10, we monitored the `redis` cache as shown in Figure 29. Analysing the needed metrics, we can see that there are no errors for any of the APIs. However, when compared to the first experiment, there is no noticeable improvement in the *response-times* as well.

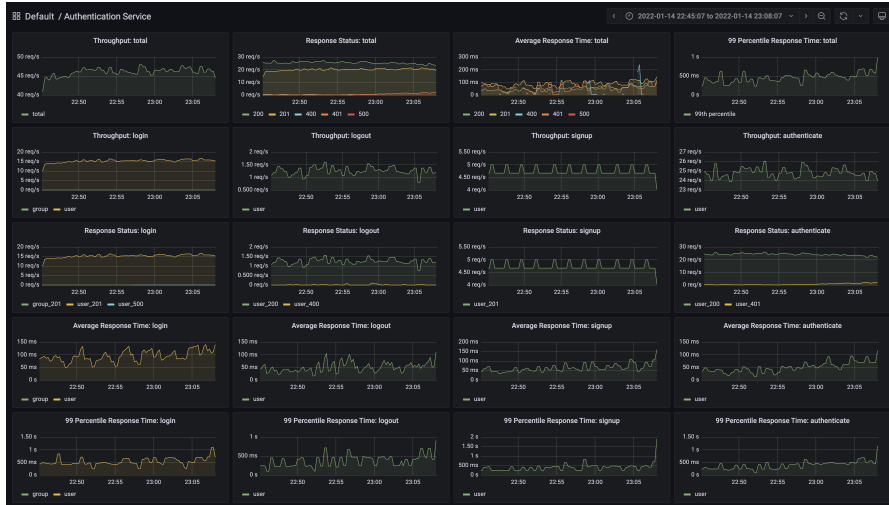


Figure 28: Authenticator Dashboard [Experiment 2]

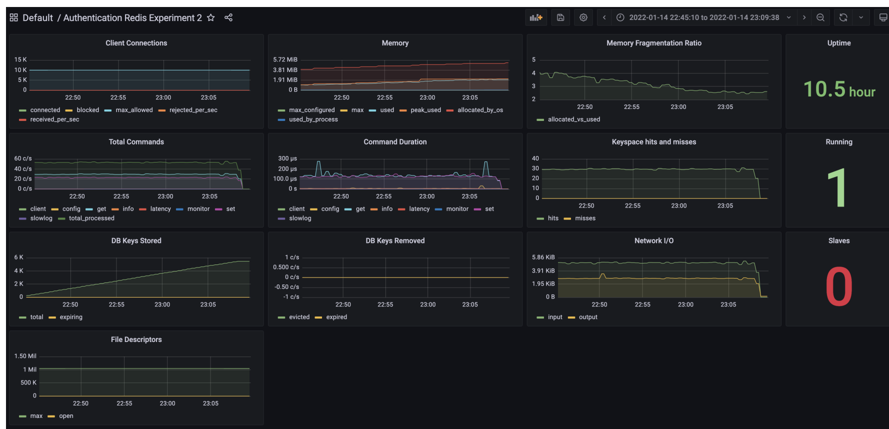


Figure 29: Authenticator Redis Dashboard [Experiment 2]

Looking into the system health metrics, we can see in Figure 30 that the *CPU Load* for the service instance is `42%` less than that of the current system and about `38.4%` less than that of the first experiment. For the database and the `redis` instances combined, however, it is `66.67%` more than the first experiment, but `25%` less than the current system. Comparing in terms of process memory usage, as Figure 33, Figure 34 and Figure 35 shown, it is less than both the previous implementations for all instances, i.e., the service, the database as well as the `redis`. Comparing the overall memory footprint of this configuration with other the two, it is `13.8%` less than the current system and `23.5%` less than first experiment. Persistent volume usage, monitored in

Figure 36, is however 12 MB higher that the current implementation and 29 MB higher than the master database of the experiment 1.

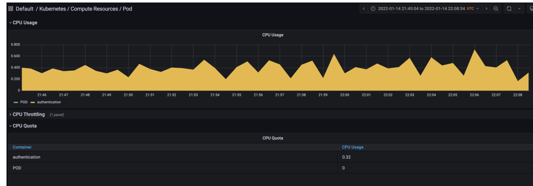


Figure 30: Authenticator Service Pod CPU Load [Experiment 2]

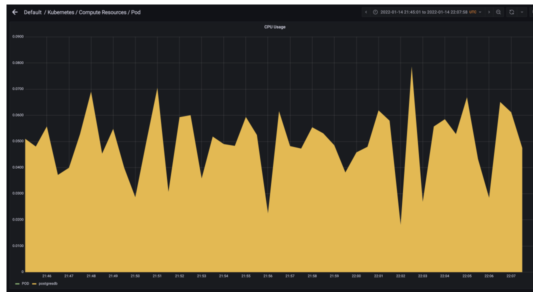


Figure 31: Authenticator DB Pod CPU Load [Experiment 2]

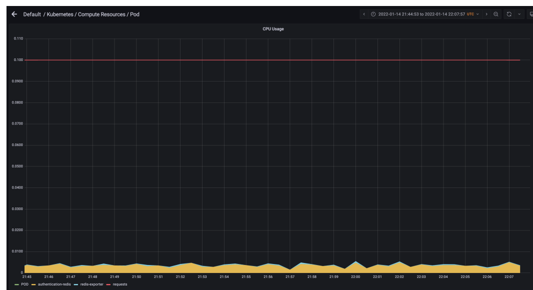


Figure 32: Authenticator Redis Pod CPU Load [Experiment 2]

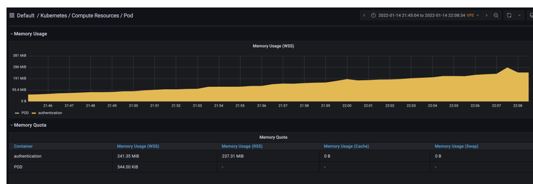


Figure 33: Authenticator Service Pod Memory Usage [Experiment 2]

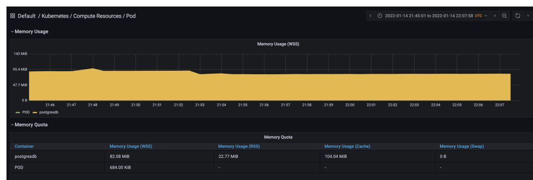


Figure 34: Authenticator DB Pod Memory Usage [Experiment 2]

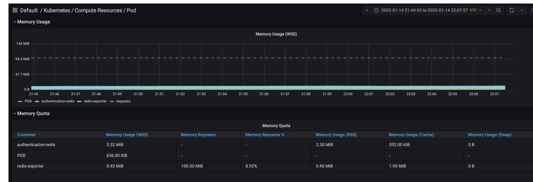


Figure 35: Authenticator Redis Pod Memory Usage [Experiment 2]

Figure 36: Authenticator DB Persistent Volume Usage [Experiment 2]

5.1.3 *Evaluate the results and compare across relevant parameters*

We summarise the results obtained from all the above system evaluations in the tables below. As per the observations made in Table 3, all the solutions provide strong consistency and high availability with *zero* server errors. However, the *master/slave* database configuration of the first experiment results in lowest response latencies from the service. However, Table 4 shows that the infrastructure for the second experiment is the most cost efficient, because it has the lowest *CPU Load* of `0.475` in total and the lowest *memory usage* of `326 MB` in total, thus leading to low infrastructure cost. The persistent volume usage of the database is the lowest for the current system, as noted in Table 5.

System Configuration	API Type	Response Time	Server Errors
Current System	Read API	`150 ms`	`0`
	Write API	`150 ms`	`0`
Experiment 1	Read API	`35 ms`	`0`
	Write API	`100 ms`	`0`
Experiment 2	Read API	`50 ms`	`0`
	Write API	`100 ms`	`0`

Table 3: Authenticator Evaluation: API Response Times and Server Errors

System Configuration	Instance	CPU Load	Memory Usage
Current System	Service	0.700	325 MB
	DB	0.100	53 MB
Experiment 1	Read Service	0.200	52 MB
	Write Service	0.450	278 MB
	Master DB	0.020	66 MB
	Slave DB	0.025	30 MB
Experiment 2	Service	0.400	241 MB
	DB	0.070	82 MB
	Redis	0.005	3 MB

Table 4: Authenticator Evaluation: System Health Metrics

System Configuration	Instance Type	Persistent Volume Usage
Current System	DB	135 MB
Experiment 1	Master DB	118 MB
	Slave DB	102 MB
Experiment 2	DB	147 MB

Table 5: Authenticator Evaluation: Persistent Volume Usage

The decision about the most suited configuration here highly depends on the preferred order of priority between response latency and estimated infrastructure expense. In terms of infrastructure cost differences, all the configurations have almost similar overall *CPU Load*. They differ in their memory and persistent volume usage. Current infrastructure provisions `4 GB` instances, and uses `gp2 SSD` disks for persistent storage which are priced at `$ 0.11` per GB per hour. Both the experiments provide same write API latency, and differ only `15 ms` in the read API latency. However, the memory requirement is almost double for the first experiment. Thus, if infrastructure cost is a big factor and a delay of `15 ms` is tolerable on the read API, we recommend the second experiment for this microservice.

5.2 FILE UPLOADER

The load testing script for the `file-uploader` fired the *file-upload* API with a request rate of `20` requests per second for `30` minutes. Each of these requests simulated the file upload action initiated by the user, by uploading a file of `944 KB`. Let us see how each of the system configurations reacted to this test. An important assumption made in here is that all the other microservices in the BranchKey platform perform to their ideal capacity. A highly optimal `api-gateway` is assumed with minimal addition to the response latency for the *file-upload* API calls. `central-aggregator` is assumed to be fast enough that the data in `redis` need not be stored for more than `5` seconds.

5.2.1 *Current System*

We prepared a *grafana* dashboard as shown in Figure 37 to monitor the service level metrics for the `file-uploader`. A similar dashboard to monitor the health and performance of the `redis` was also provisioned, as shown in Figure 38.

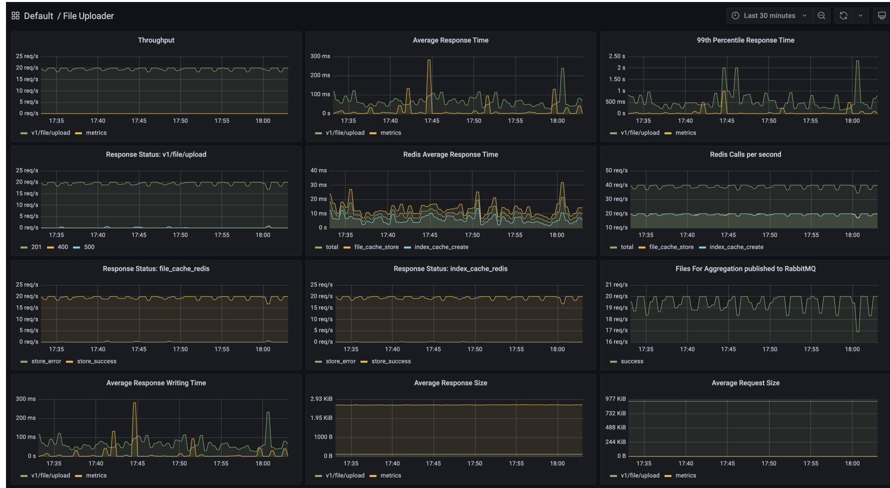


Figure 37: File Uploader Dashboard [Current System]

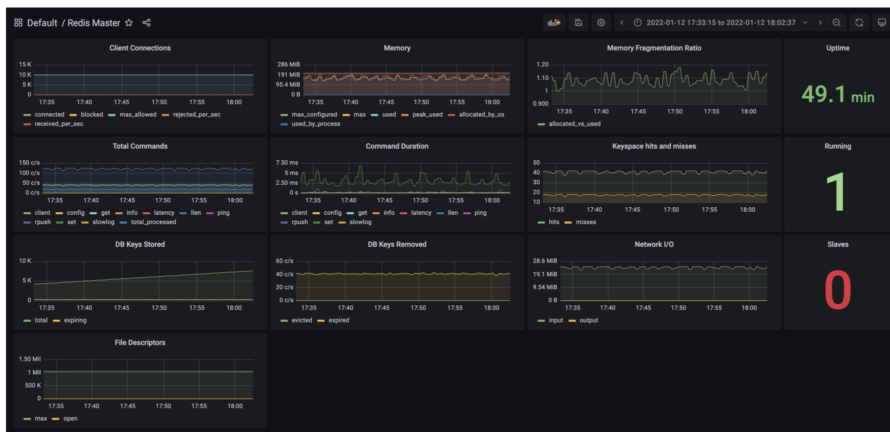


Figure 38: File Uploader Redis Dashboard [Current System]

As can be seen in Figure 37, the system was subjected to a load of about *20 requests per second*. Since every API request leads to 2 `redis` calls, the number of `redis` calls per second is `40`. Intermittent dips in the graph can be attributed to a temporary sleep window configured in the load tester, so as to keep a check on the number of connections and resources being used.

Magnifying into the relevant metrics, we first dive into the *response-time* of the service and the cache. The overall time taken by the service to analyse and processing the incoming request body, calling `redis` for storing the data, and finally formulating the response for the client is averaged over all the calls to plot as *Average Response Time* in Figure 37. This value rises to

a maximum of `125` milliseconds(ms).[1] In here, the end-to-end call from the service to the `redis` takes less time, as shown in the plot for *Redis Average Response Time*. It is about `20 ms` for storing a file, where as only *10 ms* for storing the index data. This makes a total of approximately `30 ms` for the `redis` calls, which means the domain layer handling of the request takes `90 ms` for this Golang application. Another thing to note here is that a command duration, as experienced by `redis`, is even less. It is only `5 ms` for a `SET` command, as shown in the plot for *Command Duration* in Figure 38. We can attribute a difference of `15 ms` experienced by the service for this call to network latency.

Looking into the error rate of the system in Figure 37, *Response Status: v1/file/upload* plots the response status of the API calls against the time interval, and *Response Status: file_cache_redis* and *Response Status: index_cache_redis* show the error rate for the corresponding `redis` calls. As we can see in all these graphs, most calls succeed. Error is rate is very low for all of these.

Finally, monitoring the system health of the service pod as well as the `redis` pod, we observe in Figure 39 and Figure 40 that with this throughput, the current deployments are manageable in terms of *CPU load*, with a value of `0.25` and `0.10` respectively. As far as memory is concerned, service is only taking up about `17.5 MB` (Figure 41 and the `redis` is limited to about `177.4 MB` (Figure 42), with memory fragmentation ration less than `1.15`, as seen in *Memory Fragmentation Ratio* in Figure 38.

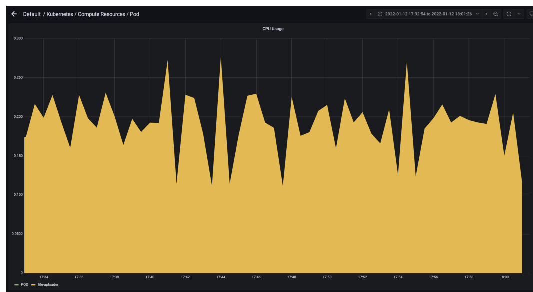


Figure 39: File Uploader Service Pod CPU Load [Current System]

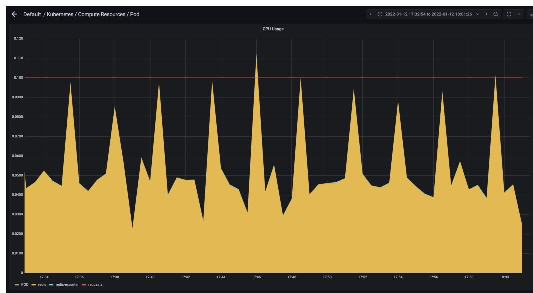


Figure 40: File Uploader Redis Pod CPU Load [Current System]

1 An exceptional peak seen can be attributed to some network delay experienced by the system, beyond the control of application logic.

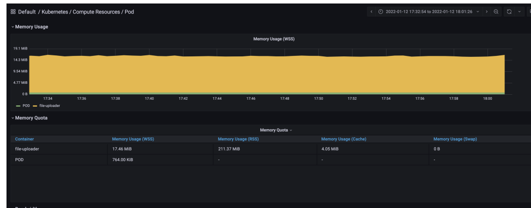


Figure 41: File Uploader Service Pod Memory Usage [Current System]

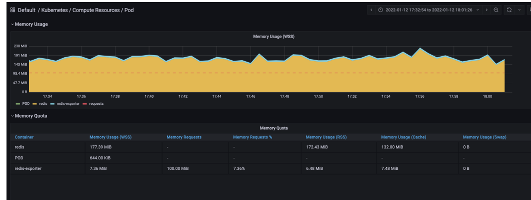


Figure 42: File Uploader Redis Pod Memory Usage [Current System]

5.2.2 *Results from the experiments*

Let us now see the observations made from the two experiments performed over this microservice.

5.2.2.1 *Experiment 1*

The first experiment for `file-uploader` tweaks the `redis` to add a data persistence and slave replication. As described in **FileUploader.1:EP.5**, we initially configured *RDB* + *AOF* persistence for `redis`. *RDB* allows for an asynchronous data dump into a persistent file at regular intervals[2]. *AOF* synchronously creates a log of all the write commands and replays them in case of failures. Although the files to be stored in `file-uploader` are big in size (`1 MB`) but only need to be stored for a small period (5 seconds). Thus, even though `redis` keys were expiring continuously keeping the memory usage in check, the *AOF* file kept growing beyond feasibility. Thus, this approach was modified to only use *RDB* persistence[3].

Thus, subjecting the above described system to the stipulated load testing script, we observed the following dashboards for the service in Figure 43, and the *master* and the *slave* `redis` in Figure 44 and Figure 45

2 Configured to happen every `300` seconds if at least `1` key is changed
3 However, with increased efficiency. Now, a backup was taken every `5` seconds if at least `1` key is changed.

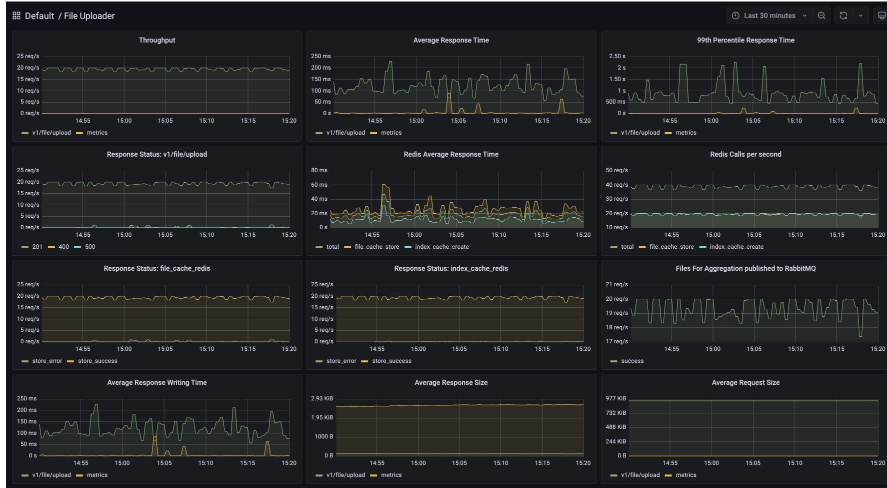


Figure 43: File Uploader Dashboard [Experiment 1]

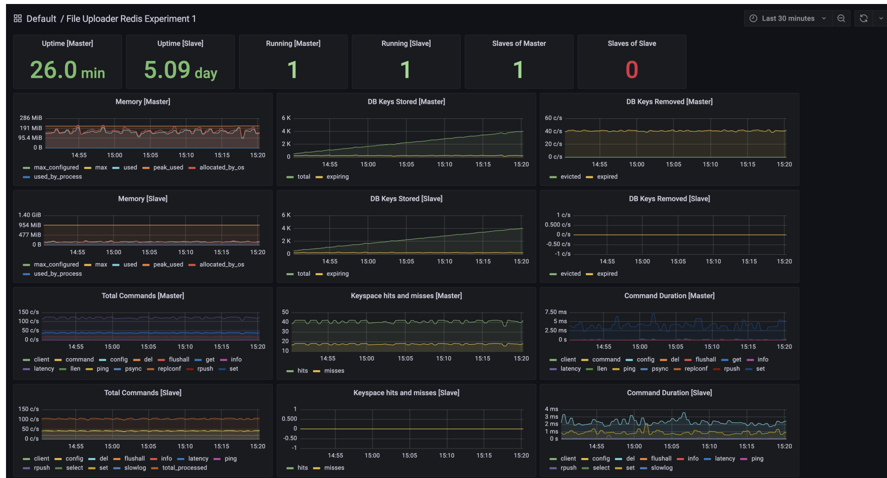


Figure 44: File Uploader Redis Dashboard [Experiment 1]

Figure 45: File Uploader Redis Dashboard (Continued) [Experiment 1]

As we can see in *Throughput* and *Redis Calls Per Sec* Figure 43, service throughput is maintained at `20` *requests per second* with `40` *calls per second* for `redis`.

Observing the *response-time* for this load, we can see in *Average Response Time* in Figure 43 that the service response time increases to about `200 ms`, with a corresponding increase in response from `redis` as well. As plotted in *Redis Average Response Time*, the time for storing a file increases to about `25 ms` and that for index data increases to about `15 ms`. However, the increment in the command duration, plotted as *Command Duration* in Figure 44, is only to `7.13 ms`. This means that `redis` takes an extra `5 ms` to perform the synchronous slave replication. However, the service experiences an extra `75 ms` of delay. This could be because of asynchronous data persistence happening in the `redis`, thus affecting its availability and latency.

Exploring the error rates in this setup, we can see the service and the redis error rates in Figure 43 and the `redis`. There is a slight increase in the number of `500` errors from the service, mainly propagated by the errors from the *file-store* call on *redis*.

Lastly, evaluating the effects of this configuration over the system health, we see in Figure 46, Figure 47 and Figure 48 that the *CPU Load* on the service remains the same as the current system. However, the *CPU Load* for the `redis` pods has increased as compared to the current system implementation, with a total summed up load of `0.45`. However, this value is still far from alarming levels. In terms of memory usage, service pod is now using an increased amount of about `23.7 MB` Figure 49, where as the `redis` is now using a little lesser memory of `156 MB` Figure 50 with a similar memory fragmentation of `1.15`, as shown in Figure 45. Slight reduction of `12.4%` in the memory usage of `redis` could be due to continuously expiring keys, and thus can be ignored for practical purposes.

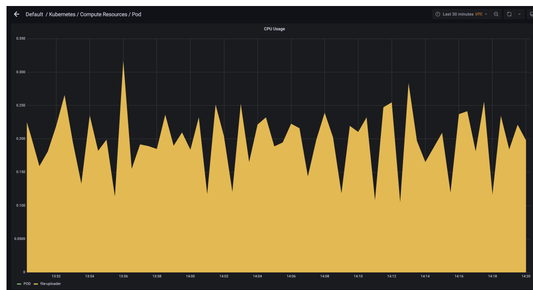


Figure 46: File Uploader Service Pod CPU Load [Experiment 1]

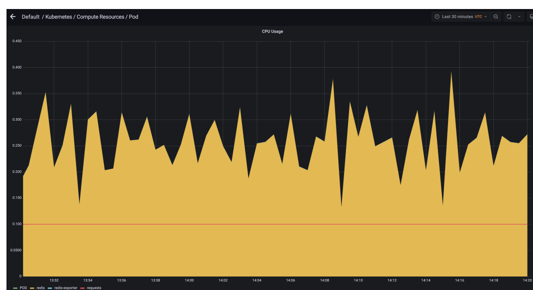


Figure 47: File Uploader Redis Master Pod CPU Load [Experiment 1]

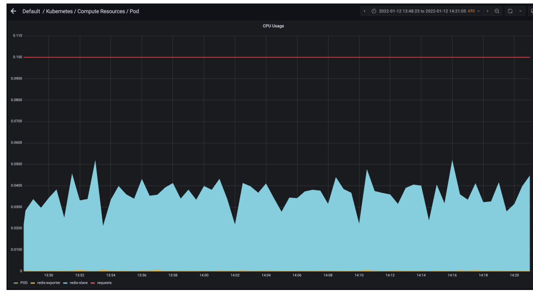


Figure 48: File Uploader Redis Slave Pod CPU Load [Experiment 1]

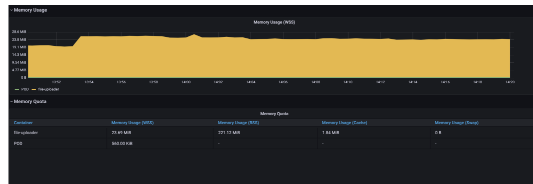


Figure 49: File Uploader Service Pod Memory Usage [Experiment 1]

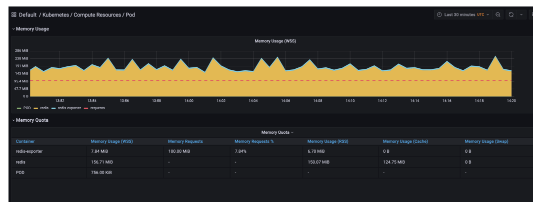


Figure 50: File Uploader Redis Master Pod Memory Usage [Experiment 1]

Figure 51: File Uploader Redis Persistent Volume Usage [Experiment 1]

5.2.2.2 *Experiment 2*

In this experiment we explore horizontal partitioning of file data by sharding it across multiple `redis` nodes. We employ a *redis-cluster* of `6` nodes, with a configuration of `3` *masters* and `3` *slaves*. Additionally, we add persistence to the data stored therein. However, as discussed in the previous experiment, we use only *RDB* persistence.[4]

Applying the load of `20` *requests per second*, each with a file of size `1 MB`, this system produced the following dashboard for service (Figure 52). For evaluating the `redis` instances, instead of observing all the `6` instances, we randomly pick out one *master* and one *slave* instance. Their respective dashboards are shown in Figure 53 and Figure 54.

4 With the same configuration as the one in Experiment 1.

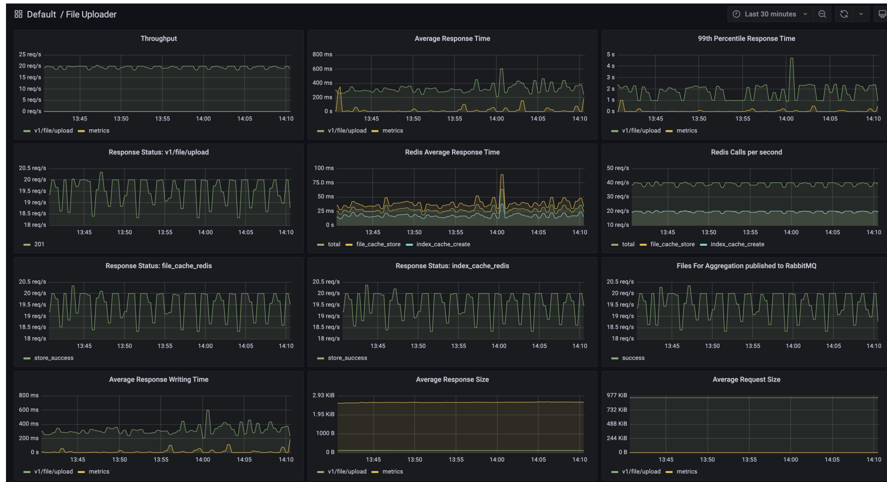


Figure 52: File Uploader Dashboard [Experiment 2]

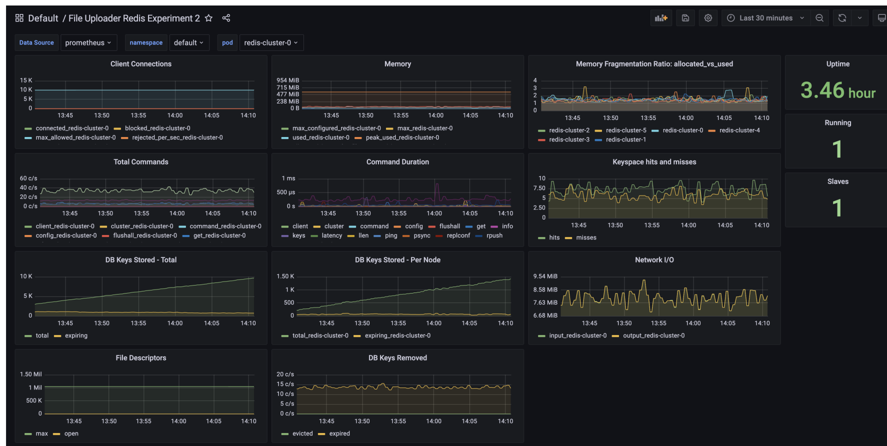


Figure 53: File Uploader Redis Master Dashboard [Experiment 2]

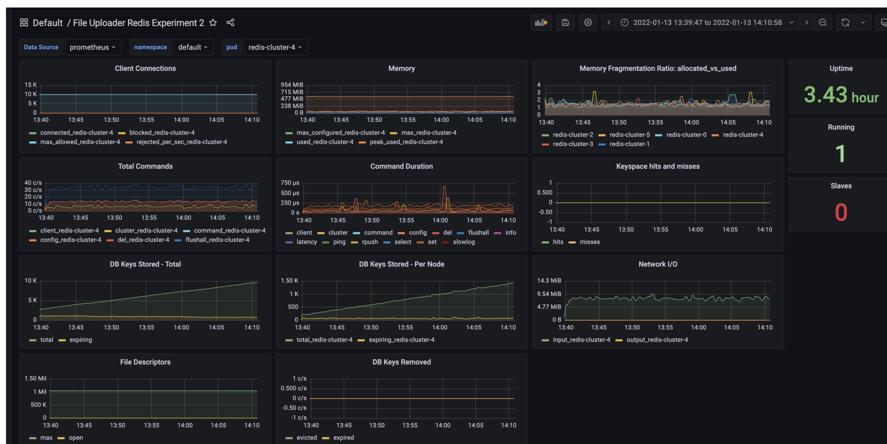


Figure 54: File Uploader Redis Slave Dashboard [Experiment 2]

As is evident in *Throughput* and *Redis Calls Per Sec* in Figure 52, service throughput is maintained at `20` *requests per second*, thus subjecting `redis` to `40` *calls per second*.

As expected, the API *response-time* for this setup has increased to an average value of 350 ms even though the increment in redis response time is only to 40 ms for storing file and 25 ms for storing the index data. This is because the time to communicate with a single redis instance is still the same, however, the cluster configuration sometimes demands communication with multiple instances. This happens especially when the needed key is not stored in the communicated instance. Although, the sharding of dataset has reduced the lookup time through each of the instances, thus reducing the command duration for each redis instance to only 3 microseconds.

Looking into the error rates for this experiment, we notice that there are absolutely no errors experienced. All the API calls return success response. This means the the cluster configuration does live up to its claim of high availability.

Finally, evaluating system health for this configuration, we see that the *CPU Load* for service (shown in Figure 55) is 0.25, which is equal to that of the current system and the experiment 1. For the redis instances, the load of master in Figure 56 is 0.12, which is less than the half of that compared from experiment 1. Similarly, the *CPU Load* for the slave instance is 0.1, as shown in Figure 57. This value is also half of the load for the *slave* instance in the first experiment. Overall, the system is healthy in terms of its computation requirements. In terms of memory usage, service pod, as shown in Figure 49, uses the same memory of about 23.8 MB as in the first experiment. However, the memory footprints of each of the redis instances, as can be seen in Figure 59 and Figure 60, has decreased drastically to 52 MB and 53 MB respectively. This is because the total memory usage is now distributed across all the nodes of the cluster. Hence, the overall memory footprint is still the same (or even more depending on the number of nodes in the cluster). A similar trend is also followed by the persistent volume usages as described in Figure 61 and Figure 62. A single *master* instance uses 61 MB. If we use this as a base value to estimate the persistent volume usage for all the three *master* nodes, it would be 183 MB, which is about 23 MB more than that of first experiment.

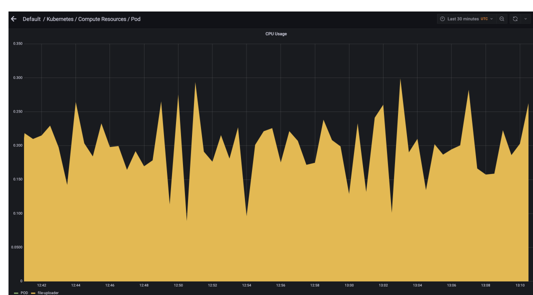


Figure 55: File Uploader Service CPU Load [Experiment 2]

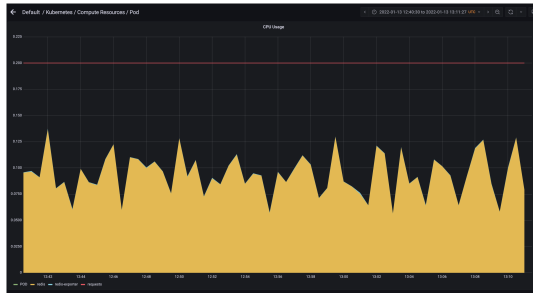


Figure 56: File Uploader Redis Master CPU Load [Experiment 2]

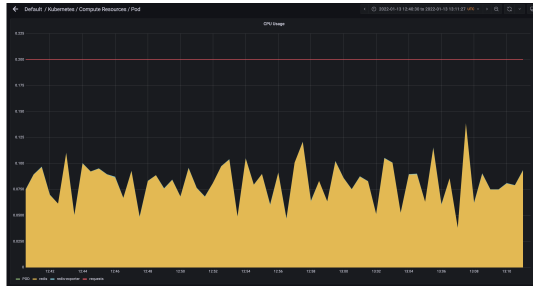


Figure 57: File Uploader Redis Slave CPU Load [Experiment 2]

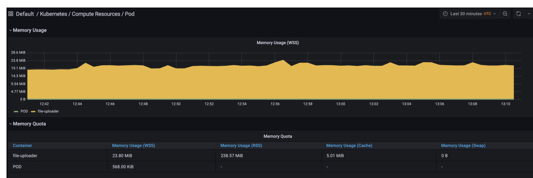


Figure 58: File Uploader Service Pod Memory Usage [Experiment 2]

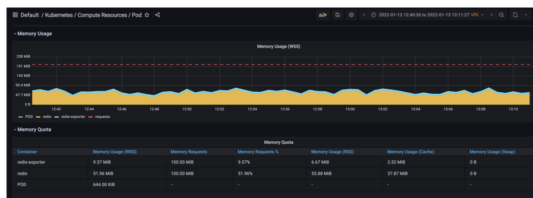


Figure 59: File Uploader Redis Master Pod Memory Usage [Experiment 2]

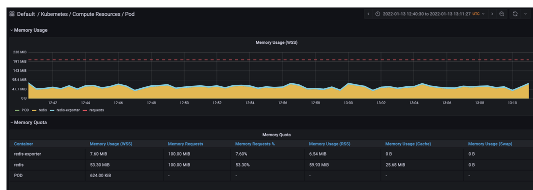


Figure 60: File Uploader Redis Slave Pod Memory Usage [Experiment 2]

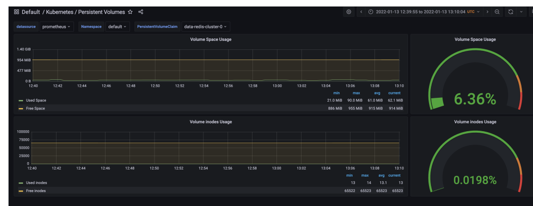


Figure 61: File Uploader Redis Master Persistent Volume Usage [Experiment 2]

Figure 62: File Uploader Redis Slave Persistent Volume Usage [Experiment 2]

5.2.3 *Evaluate the results and compare across relevant parameters*

All the results shown in the graphical form are tabulated below. Table 9 helps us evaluate the system configurations in terms of *response-times* and *error-rates*. We see that the current system gives the lowest overall response latency (`125 ms`) for the *file-upload* API. With regards to *error-rates*, the current system and the first experiment do report a small number of system errors. They are observed due to limited system resources. The high throughput from the load test makes the service or the `redis` pod reach the limit of maximum open connections allowed, or the maximum resources available, thus giving server errors to all the requests received until the engaged resources are freed up. However, the second experimental setup gives absolutely *zero* server errors, thus guaranteeing high availability. Further, we can see that the current system is also the most infrastructurally optimised solution in terms of its lowest *CPU Load* of `0.35` in total and lowest *memory usage* of `194.9 MB`, as listed down in Table 7. Important thing to note here is that for the second experiment, there are `3` *redis masters* and `3` *redis slaves*. Thus, the *CPU Load* and the *memory usage* need to be multiplied by a factor of `3` for both these instances. One major drawback of the current system is that it lacks persistent storage of data. When compared in terms of persistent volume used, Table 8 shows that first experiment is more efficient than the second, with only `160 MB` of persistent volume used. This is because the persistent volumes for the second experiment again need to be multiple by `3` to get the actual value of `183 MB` for the *master* instances and `195 MB` for the *slave* instances.

System Configuration	Entity	Response Time	Server Errors
Current System	Overall API	125 ms	0.5 per second
	Redis File Storage	20 ms	0.25 per second
	Redis Index Storage	10 ms	0
	Redis Command Duration	5 ms	N/A
Experiment 1	Overall API	200 ms	1 per second
	Redis File Storage	25 ms	1 per second
	Redis Index Storage	15 ms	0
	Redis Command Duration	7 ms	N/A
Experiment 2	Overall API	350 ms	0
	Redis File Storage	40 ms	0
	Redis Index Storage	25 ms	0
	Redis Command Duration	0.3 ms	N/A

Table 6: File Uploader Evaluation: API Response Times and Server Errors

System Configuration	Instance	CPU Load	Memory Usage	Memory Fragmentation
Current System	Service	0.250	17.5 MB	N/A
	Redis	0.100	177.4 MB	1.15
Experiment 1	Service	0.250	23.75 MB	N/A
	Master Redis	0.300	156 MB	1.15
	Slave Redis	0.250	160 MB	1.15
Experiment 2	Service	0.250	23.8 MB	N/A
	Master Redis	0.120	52 MB	1.97
	Slave Redis	0.100	53 MB	2.52

Table 7: File Uploader Evaluation: System Health Metrics

System Configuration	Instance Type	Persistent Volume Usage
Current System	Redis	N/A
Experiment 1	Master Redis	160 MB
Experiment 2	Master Redis	61 MB
	Slave Redis	65 MB

Table 8: File Uploader Evaluation: Persistent Volume Usage

Thus, for `file-uploader`, current system seems to be the most optimised version, if it is configured with a persistent volume as well. However, `file-uploader` is a *AP* system, which means high availability is of essence here. Thus, we recommend the *redis-cluster* configuration of the second experiment. It not only promises high server availability and data persistence, but also provides an easy horizontal scaling option whenever needed. Both of the other configurations use single master `redis`, which limit the scaling capabilities in times of need.

5.3 FILE DOWNLOADER

To test the resilience of the `file-downloader`, we designed a load test to fire the service with about 5 *requests per second* of each of the *upload-aggregated-file* and the *file-download* APIs. Each of these APIs included an upload or a download of a file of `944 KB`. This microservice was tested under the assumptions of an ideal performance by rest of the entities in this ecosystem. Once an event to avail the download of an *aggregated-output-result* file is sent to the clients, it is assumed that all of them download the file within `10` *seconds*. Following this assumption, we mark a file as expired after `10` *seconds* of its creation, and run a background job to periodically delete all such expired file. Thus, all the clients requesting for a file after `10` *seconds* get a `HTTP 404: file not found` error.

5.3.1 *Current System*

We subjected the current system implementation to the load specified in Section 4.3 and observed the *grafana* dashboards for the service and the database shown in Figure 63, Figure 64 and Figure 65. Here we can see that there are no `HTTP 500:service unavailable` errors for either of the API calls. Additionally, the *response-times* are bound within `30 ms` for the *upload-aggregated-file* API and within `5 ms` for the *file-download* API. Breaking this down to the database level, it takes about `5 ms` to perform a write query whereas about `2 ms` to perform a read query.

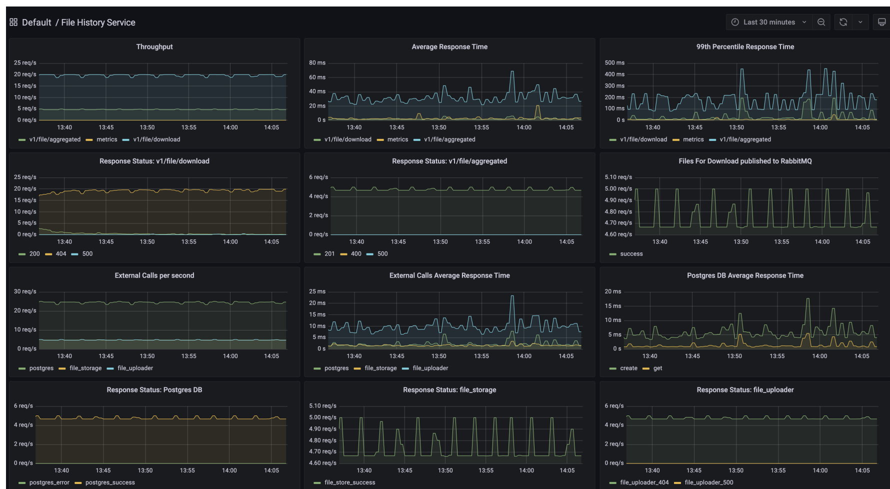


Figure 63: File Downloader Dashboard [Current System]

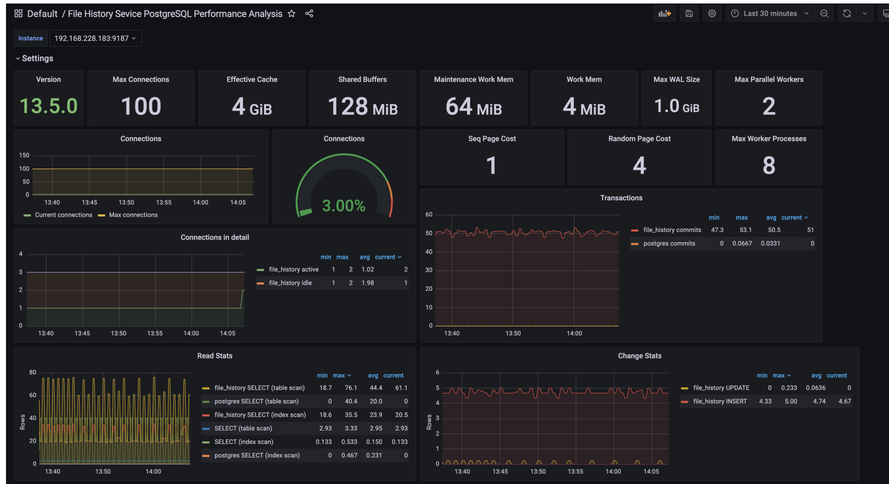


Figure 64: File Downloader PostgreSQL Dashboard [Current System]

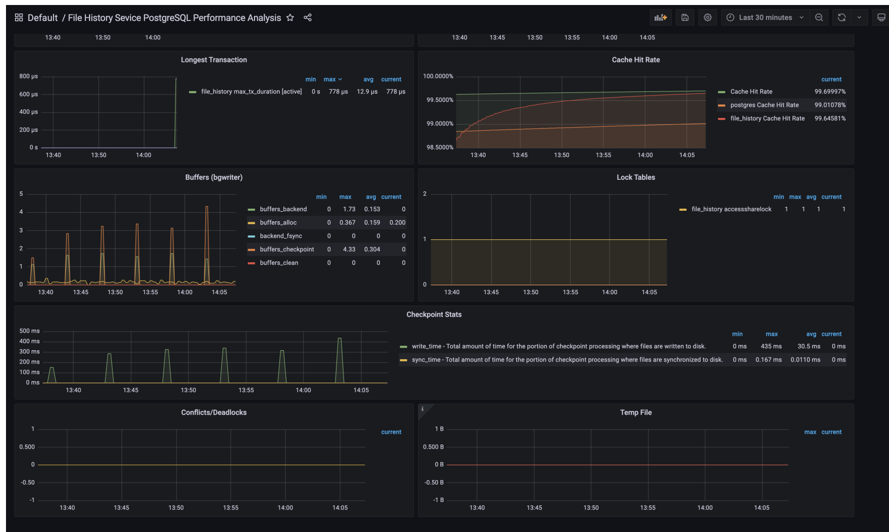


Figure 65: File Downloader PostgreSQL Dashboard (Continued) [Current System]

Assessing the system health aspects of the system, we can see in Figure 66 and Figure 67 that the *CPU Load* for both the service as well as the database instance is very low, `0.05` and `0.01` respectively. The *memory usage* is also only `23.5 MB` for the service (Figure 68) and `42.4 MB` for the database (Figure 69). Thus, the current implementation is highly optimised in terms of its infrastructural requirements. The persistent volume requirement is `103 MB` on average, as observed in Figure 70.

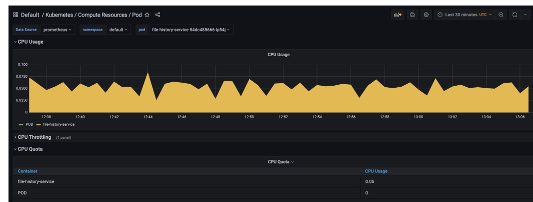


Figure 66: File Downloader Service Pod CPU Load [Current System]

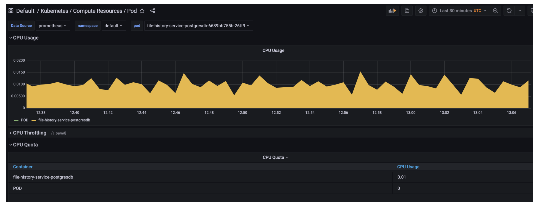


Figure 67: File Downloader DB Pod CPU Load [Current System]

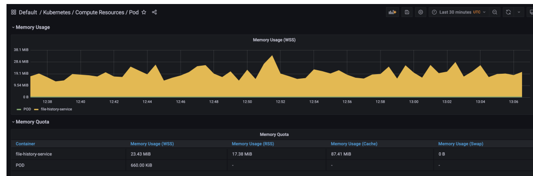


Figure 68: File Downloader Service Pod Memory Usage [Current System]

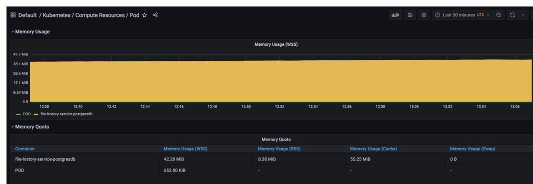


Figure 69: File Downloader DB Pod Memory Usage [Current System]

Figure 70: File Downloader DB Persistent Volume Usage [Current System]

5.3.2 *Results from each experiments*

Let us now monitor the results from the two experiments performed for this microservice.

5.3.2.1 *Experiment 1*

In this experiment, we assess the performance of the *master/slave* configuration of the database. With the logical distribution of write queries to the *master* database and the read queries to the *slave* database, we observe the service metrics as shown in Figure 71 and the database metrics as shown in Figure 72 and Figure 73. We see here that the *response-times* have in fact increased for both the APIs. The database *response-time* has increased to `10 ms` for a write query and to `5 ms` for a read query, thus leading to increased *response-times* of `40 ms` for *upload-aggregated-file* API and about `5 ms` for the *download-file* API. Error rates however, still remain *zero* for both the APIs.

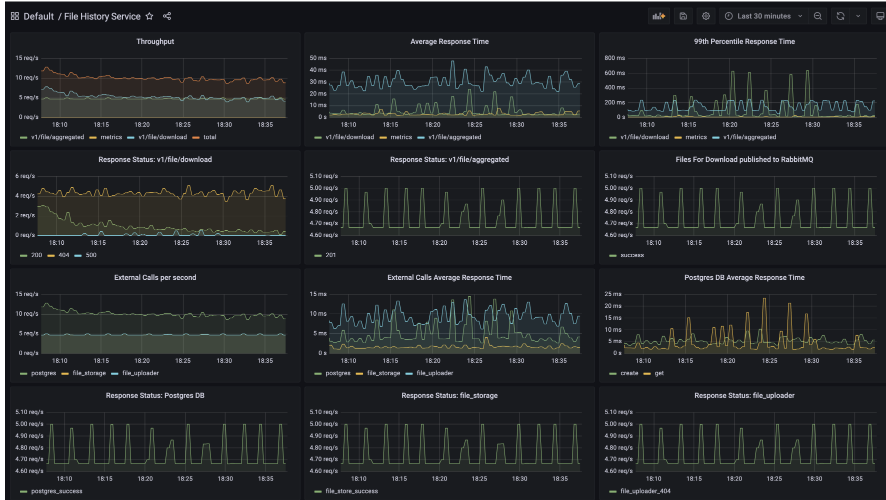


Figure 71: File Downloader Dashboard [Experiment 1]

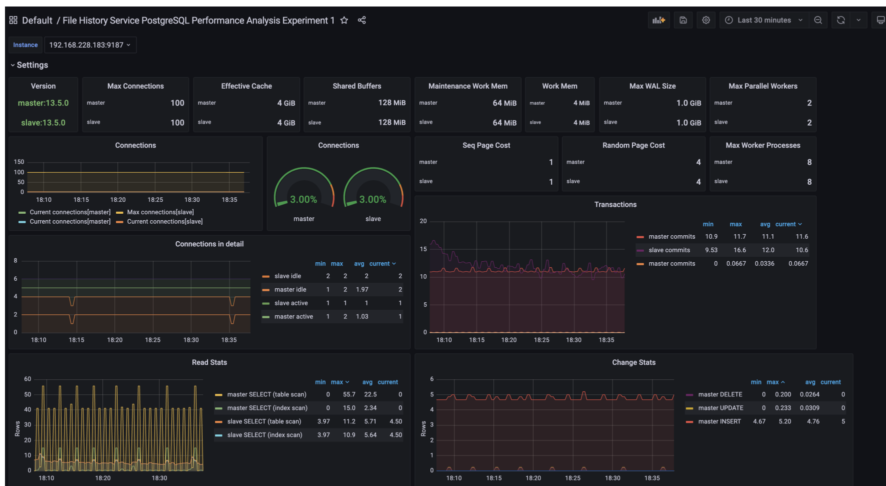


Figure 72: File Downloader PostgreSQL Dashboard [Experiment 1]

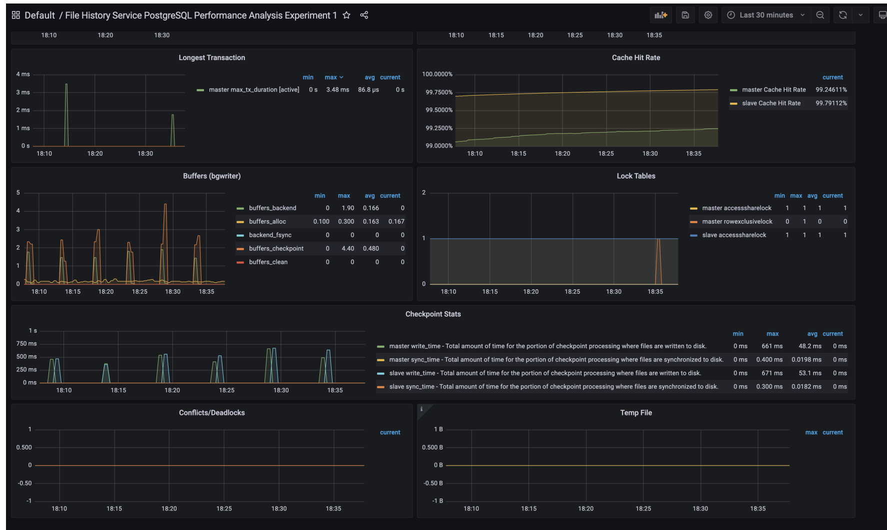


Figure 73: File Downloader PostgreSQL Dashboard (Continued) [Experiment 1]

Evaluating the system health metrics for this configuration, we see that the *CPU Load* for the service instance, shown in Figure 74 is `0.01` less than that of the current system. However, the same metric for the *master* and *slave* databases, shown in Figure 75 and Figure 76 respectively, is `0.01` more than the current implementation. In fact, both of the combined become `0.04`, which is a four times that of the current implementation. Extending the discussion to *memory-usage*, the memory requirements of the service instance, noted in Figure 77, are reduced to only `19.2 MB`. However, the memory requirements of the database instances, shown in Figure 78 and Figure 79 are increased to `52.6 MB` and `48.8 MB` respectively, as compared to the current system. The persistent volume usages, marked in Figure 80 and Figure 81, remain the same to about `103 MB` for the **master** and `102 MB` for the *slave* databases. Overall, we can say that infrastructurally, the service instance is lighter while the database instances are heavier than the current implementation.

Figure 74: File Downloader Service CPU Load [Experiment 1]

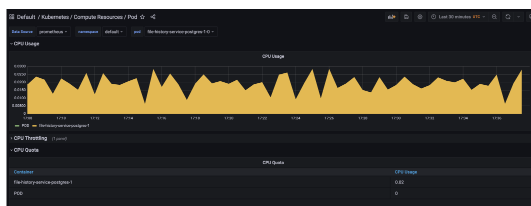


Figure 75: File Downloader Master DB CPU Load [Experiment 1]

Figure 76: File Downloader Slave DB CPU Load [Experiment 1]

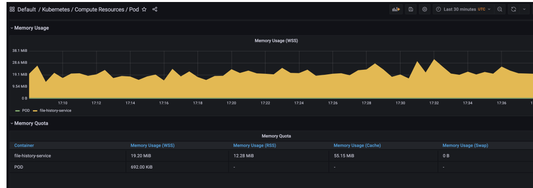


Figure 77: File Downloader Service Memory Usage [Experiment 1]

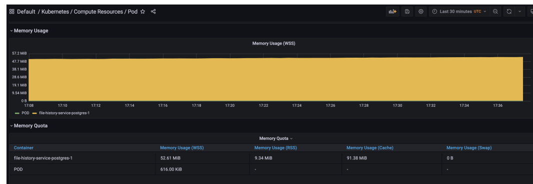


Figure 78: File Downloader Master DB Memory Usage [Experiment 1]

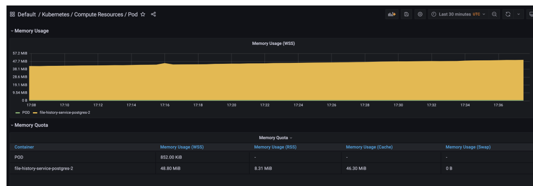


Figure 79: File Downloader Slave DB Memory Usage [Experiment 1]

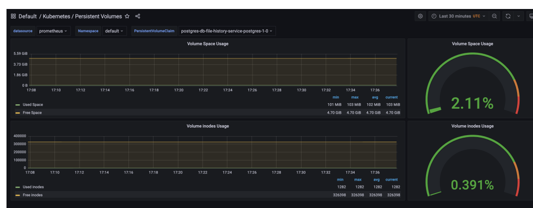


Figure 80: File Downloader Master DB Persistent Volume Usage [Experiment 1]

Figure 81: File Downloader Slave DB Persistent Volume Usage [Experiment 1]

5.3.2.2 *Experiment 2*

The second experiment for `file-downloader` explores the effects of horizontal data partitioning over the service performance. For the sake of this experiment, we perform hash partitioning over the PostgreSQL table. We create 5 partitions to hash the group-ids. We seed the database with `30` group-ids, randomly distributed over these partitions, and load test the system under such conditions. The observed system metrics are recorded in Figure 82. When comapred against the current system, we see that the *response-times* of the APIs have increased to about `10 ms` for the *file-download* API and to `40 ms` for the *upload-aggregated-file* API. This increment is observed despite a decrement of `1.5 ms` in the database response time for read queries. Service, however, still remain highly available, with no `500` errors.

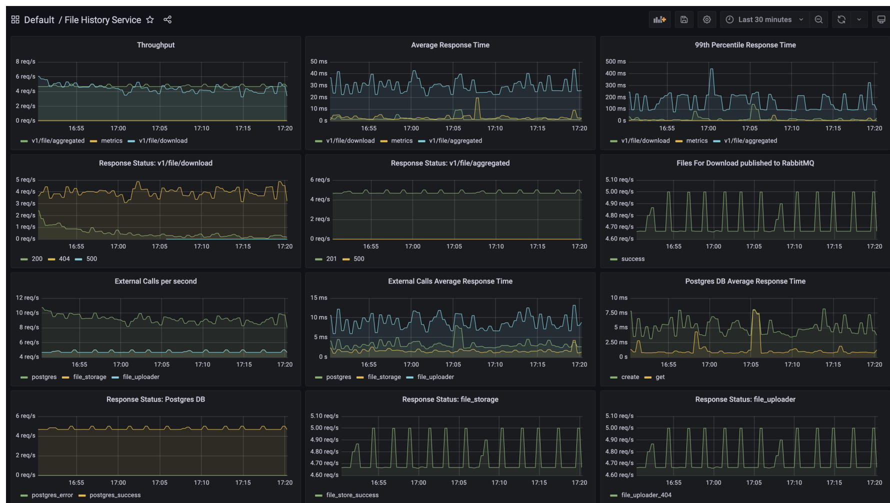


Figure 82: File Downloader Dashboard [Experiment 2]

Considering the system health metrics, Figure 83 and Figure 84 show that the *CPU Load* for the service as well as the database instances remain the same as the current system, to values of `0.05` and `0.01` respectively. The *memory-usage* however has decreased to `19.9 MB` for the service and `27.3 MB` for the database, as shown in Figure 85 and Figure 86 respectively. The average persistent volume usage of the system, as monitored in Figure 87, has increased slightly to *107 MB*.

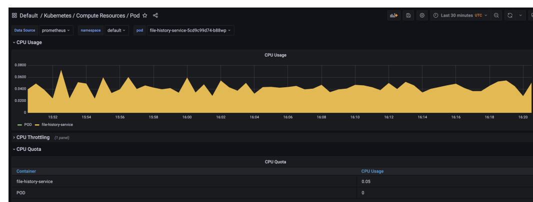


Figure 83: File Downloader Service Pod CPU Load [Experiment 2]

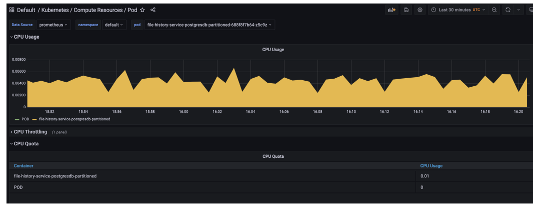


Figure 84: File Downloader DB Pod CPU Load [Experiment 2]

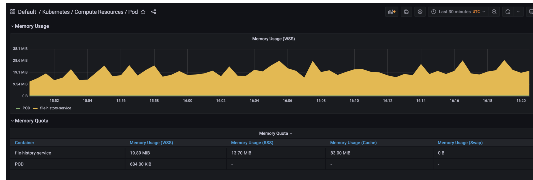


Figure 85: File Downloader Service Pod Memory Usage [Experiment 2]

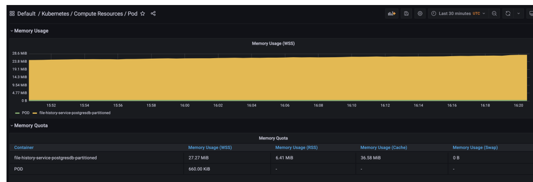


Figure 86: File Downloader DB Pod Memory Usage [Experiment 2]

Figure 87: File Downloader DB Persistent Volume Memory Usage [Experiment 2]

5.3.3 *Evaluate the results and compare across relevant parameters*

This section summarises the results obtained from all the above experiments on `file-downloader`, and compares them against the current system configuration. Table 9 provides a tabular comparison of the three system configurations in terms of *response-times* and *error-rates*. Here, we see that the current system performs the best for the given load requirements. Although all the system configurations have *zero* server errors, thus guaranteeing high server availability, the current system responds with the lowest latencies of `30 ms` and `5 ms` for both the APIs respectively. In terms of infrastructural parameters seen in Table 10, all the systems have nearly similar total *CPU Load* requirements. The second experiment however, uses the least memory of `47.2 MB`. In terms of persistent volume storage, the current system is the most efficient with the least usage of `103 MB`, as can be seen in Table 11. The persistent volume usage of the second experiment is only `5 MB` more than the current system, where as for the first experiment, it is nearly double.

System Configuration	Entity	Response Time	Server Errors
Current System	Upload Aggregated File API	30 ms	0
	Download File API	5 ms	0
	DB Write	5 ms	0
	DB Read	2.5 ms	0
Experiment 1	Upload Aggregated File API	40 ms	0
	Download File API	5 ms	0
	DB Write	10 ms	0
	DB Read	5 ms	0
Experiment 2	Upload Aggregated File API	40 ms	0
	Download File API	10 ms	0
	DB Write	6.3 ms	0
	DB Read	1 ms	0

Table 9: File Downloader Evaluation: API Response Times and Server Errors

System Configuration	Instance	CPU Load	Memory Usage
Current System	Service	0.050	23.5 MB
	DB	0.010	42.4 MB
Experiment 1	Service	0.040	19.2 MB
	Master DB	0.020	52.6 MB
	Slave DB	0.020	48.8 MB
Experiment 2	Service	0.050	19.9 MB
	DB	0.010	27.3 MB

Table 10: File Downloader Evaluation: System Health Metrics

System Configuration	Instance Type	Persistent Volume Usage
Current System	DB	103 MB
Experiment 1	Master DB	103 MB
	Slave DB	102 MB
Experiment 2	DB	107 MB

Table 11: File Downloader Evaluation: Persistent Volume Usage

Even though the proposed solutions promise better scalability and fault tolerance, we recommend the current system configuration for the current requirements of the `file-downloader`. For now, both of the other solutions seem over-engineered for the system. However, in case horizontal scalability is needed in near future, we recommend the horizontally partitioned database schema employed in the second experiment.

5.4 DISCUSSION

Let us now reflect upon the results of all these experimental evaluations in terms of the research questions presented in Section 1.2. This research focuses on helping the developers to optimize a data-focused microservice application impacted by CAP theorem, in terms of architecture and infrastructure. This aim is highly abstract in nature, and requires immense tangibility. To effectuate it empirically, we break it down into three sub-questions **RQ.1**, **RQ.2** and **RQ.3**, each with a substantial goal to be achieved. The literature review performed in Chapter 2, and the research framework described in Chapter 4 attempt to answer each of these sub-questions. All of these sub-questions are individually answered with respect to the specific requirements of the microservice under investigation.

RQ.1 emphasises the restrictions imposed by the CAP theorem limitations over the available design choices for architectural and infrastructural solutions of a data-focused microservice in general. Since we limit our project to data-focused microservice applications, infrastructural solutions mainly cater to database solutions. Tracing this into our research, Section 2.2 and Section 2.3 provide an extensive analysis of the database patterns and the corresponding infrastructural solutions for each of the *CP*, *AP* and the *CA* systems. Further, Section 2.4, Section 2.5 and Section 2.6 review the existing literature in detail to correlate the architectural patterns for microservices architectures with the CAP theorem restrictions. The idea behind all of this research is to study the segregation of design solutions in accordance with the CAP theorem limitations, and compile a list of the same, to be used for optimally (re)designing a microservice belonging to either of the *CP*, *AP* and the *CA* scenarios.

RQ.2 underlines the identification of different design choices for the microservice under investigation, in terms of the architecture and infrastructure to be used. **RF.1** firstly puts down all the functional and non-functional requirements, which are used further ahead in the research to answer multiple questions. For instance, the functional requirements are translated into answering questions like whether this microservice needs a relational database or a non-relational database. Non-functional requirements acknowledge the CAP requirements of the system, the load and throughput requirements, or even the acceptable infrastructural costs for the system deployment. The **RF.2** of the research framework evaluates the requirements from **RF.1** in terms of consistency, availability and partition tolerance. This sets a context for reviewing the literature in terms of architectural patterns in **RF.3**. As illustrated in our case studies in 4, this review is then performed for either of a *CA*, *AP* or *CP* system. Important thing to note here is that the list of solutions compiled as a result of **RQ.1** is used to perform this review. Finally, the architectural patterns discovered from the literature are again classified in terms of there feasibility to be listed in **RF.5**, thus paving way for objective experiments in **RF.6**.

Further, **RQ.2** stresses the importance of infrastructural enhancement of the microservice under investigation. In context of our project, this refers to database patterns and infrastructural optimisations, mainly considered in terms of the CAP requirements of the system. Tracing this into the research framework, **RF.4** allows the researcher to study the existing literature and list down the feasible database solutions in **RF.5**. This literature review is

narrowed down in regards to the database requirements inferred from the requirements listed in RF.1. RF.4 ensures a well surveyed research in terms of the most suitable databases with a final consideration of only the feasible[5] solutions. Although this literature survey is expected to be detailed enough to consider and subjectively evaluate many different solutions, the classification into feasible solutions limits the scope of the research within practical boundaries.

Finally, RQ.3 necessitates an objective comparison of all the feasible solutions gathered in the process of answering RQ.2. This comparison is performed on different quantifiable parameters listed in EP.1. These quantifiable parameters are deduced from the non-functional requirements in RF.1 and the CAP requirements in RF.2. They are expected to compare different feasible solutions not only in terms of overall system consistency, availability and partition tolerance, but also in terms of the infrastructural costs of deployment like the computation power needed and memory used. RF.6 describes an elaborate experimentation protocol for this purpose. As described in this chapter, RF.7 provisions factual evaluations of all the feasible solutions and leads to well researched conclusions given in Chapter 6.

Consolidating the entire process, the literature review of the Chapter 2 answers the RQ.1, while the research framework and the experimentation protocol of the Chapter 4 provide answers for RQ.2 and RQ.3 respectively. Thus, looking at the bigger picture, attacking each of the individual subquestions finally leads us to answer the main research question. Combining the results obtained from RQ.1, RQ.2 and RQ.3, we eventually arrive at a well researched and pragmatically evaluated survey of different architectural designs for a data-focused microservice, impacted by the CAP theorem limitations. This enables us to heuristically decide on the best suited architectural and infrastructural strategy for the microservice under investigation.

The literature reviewed in Chapter 2 mostly depicts that the evaluation of architectural and infrastructural solutions has been a subjective domain. The research framework presented in this project however, bridges this into the domain of practical implementations and factual observations by presenting a concrete 7 step process of being able to implement and compare these architectural and infrastructural solutions objectively. Although we instantiated the whole process with BranchKey as a case study, the described research framework is generic enough to be able to be applied on any data-focused application in a microservice architecture.

5 A solution is considered feasible if it satisfies the needs of the system, has an acceptable infrastructure cost estimation, and can be implemented within project scope.

6 CONCLUSION AND FUTURE WORK

In this chapter, we discuss the conclusions drawn from this research, and list down the possibilities for future work in this field.

6.1 CONCLUSION

The aim of this project was to facilitate the evaluation of different architectural and infrastructural solutions for a data-focused microservices architecture, to achieve an optimal cost and performance trade-off. Here, infrastructure solutions are scoped out to consider only database patterns. The prime assumption here is that microservices follow the rules of a distributed system, and thus adhere to the limitations defined by the CAP theorem. As described in Section 1.2, the goal of the overall project is divided into three smaller sub-goals.

- First one aims at studying the effect of CAP theorem limitations over the architectural and database design of a data-focused microservice.
- Second sub-goal details out the process to identify different feasible architectural and database design considerations to (re)design the microservice under investigation for the specified business requirements of cost and performance.
- Third sub-goal aims at evaluating the selected database and the architectural/infrastructural designs in terms of the operational costs and performance metrics parameters. These parameters include the cost for infrastructure deployment, development and maintenance costs, and the service response latencies and error percentages.

Important thing to note here is that we do not invent new architectural or infrastructural solutions. We examine the existing literature for this purpose. To accomplish this, Chapter 2 provides a detailed analysis of the literature considered for this research. We have segregated out literature review into multiple categories ranging from the theoretical definitions and implications of the CAP theorem to the practical implementations and repercussions of applying the CAP theorem into the databases and microservices architecture.

This research is performed in collaboration with BranchKey, a *federated learning as a service* platform. Chapter 3 explains the current architecture of the BranchKey system, and describes in detail the functional and non-functional requirements of the following three microservices selected as the case studies for this research.

- `Authenticator`: It is a *CA* system with relational database.
- `File Uploader`: This is a *AP* system with key-value store.
- `File Downloader`: This is a *CP* system with a relational database as well as a distributed file storage.

This project is however, not limited to solve for these systems. It intends to provide a generic research framework to evaluate alternative designs for any data-focused microservice towards identifying the optimal architectural and infrastructural solution. Thus, in Chapter 4, we provide an elaborate but easy to follow 7 step research framework for the same. Further, we also illustrate its application over the three aforementioned microservices from the BranchKey system. Continuing over the same illustration, Chapter 5 presents a detailed overview of the results obtained from the application of the research framework over the microservices under investigation, and evaluates the results.

Discussing the specifics, and summarising the evaluations, we can make the following conclusions with respect to the specific microservices under investigation:

- For the `authenticator`, *server-side caching* is the recommended solution. It balances out well between the response latencies and the infrastructure costs. It is not as slow as the current system, and not as infrastructurally heavy as the *master/slave* architecture.
- In case of `file-uploader`, although the current system seems to be the most efficient design across all parameters, it severely lacks high availability and easy scalability. Since `file-uploader` is to be an *AP* system, we recommend the *redis-cluster* configuration with *RDB* persistence. This configuration also eases the efforts for scalability, in case our assumptions of quick data expiry are to be given upon.
- With respect to `file-downloader`, even though the *master/slave* architecture of first experiment and the *horizontally partitioned database schema* of the second experiment seem highly promising and scalable, they appear unnecessary at this point. Even though they do not add much to the response latencies, their infrastructural expenses are much greater than the current solution. They are definitely the options to look for in case this system needs to be upscaled, but for the current requirements, such a heavy infrastructure adds more cost without adding enough value to the system.

Reviewing in a general sense, this research framework proved to be an efficient facilitator in evaluating the trade-offs for systems lying on all the three edges of the CAP theorem spectrum. It takes into account the existing literature for the particular use case, and provides an experimentation protocol to test the pragmatic feasibility of the solutions suggested in the literature. This makes this whole investigation easy to replicate and applicable on any data intensive microservice. It ensures the evaluation to be well supported by both, theoretical literature as well as practical implementation. It provides examples to contemplate the observations made, and evaluate the underlying trade-offs to balance out different parameters. It provides a concrete methodology to arrive at an optimal solution for the system under investigation, with the provided requirements and assumptions. It also helps the researcher be informed about the shortcomings of the selected solutions, and keeps space of alternate designs. As a conclusive remark, it will not be wrong to say that this research lays down a tangible procedure to perform architectural and infrastructural evaluations for a data-focused microservice application.

6.2 FUTURE WORK

As comprehensive as this research sounds, it does have its own limitations, which can be worked upon in the future. Firstly, all the experiments designed and conclusions drawn herewith are a result of the literature review performed in Chapter 2. Although we tried to make this as extensive as possible, covering all the literature is beyond our scope. We may have missed out some important publications which could have changed the course of our conclusions entirely. Additionally, we could not find much literature discussing the architectural designs for microservices demanding strong consistency and partition tolerance. Although there were many solutions proposed for *AP* systems, *CP* systems were found to be highly missing. Similar observation was also made for the comparison of different relational databases. Our case studies used PostgreSQL database for storing tabular data. Evaluating alternative database management systems was a natural course of action, however, we could not find literature to list and compare them.

Secondly, the experiments performed in this project were scoped to small changes in the existing system architecture. There were solutions which required bigger changes, like evaluating Golang versus Python implementation of `authenticator` or comparing the effects of different event queues for `file-uploader`. These seemed to have promising results, but were scoped out of the current research. Developers at BranchKey may continue this research with such solutions, and arrive at better conclusions.

Thirdly, infrastructure costs for all the systems were only evaluated in terms of the current BranchKey deployment, i.e., on AWS cloud platform. There are many cloud service providers in the market with competing prices. BranchKey could also deploy their own infrastructure over an in-house data center. This research could further be extended to compare the infrastructure cost of all such solutions.

Lastly, the evaluation of different solutions in this research only considers service performance parameters and infrastructure deployment costs. It misses out on development and maintenance costs like developer hours. Future researchers could also extend the framework proposed herein to include such costs.

Part I

APPENDIX

A REQUIREMENT ANALYSIS OF OTHER BRANCHKEY MICROSERVICES

A.1 AUTHORISER

`Authoriser` is mainly responsible to store the customer authorisation configuration per group-id basis. It provides interfaces to create such records, update them and fetch them for a given group-id. Figure 88 shows the exposed interfaces for this module. It does not have any dependency on any other microservice in the system.

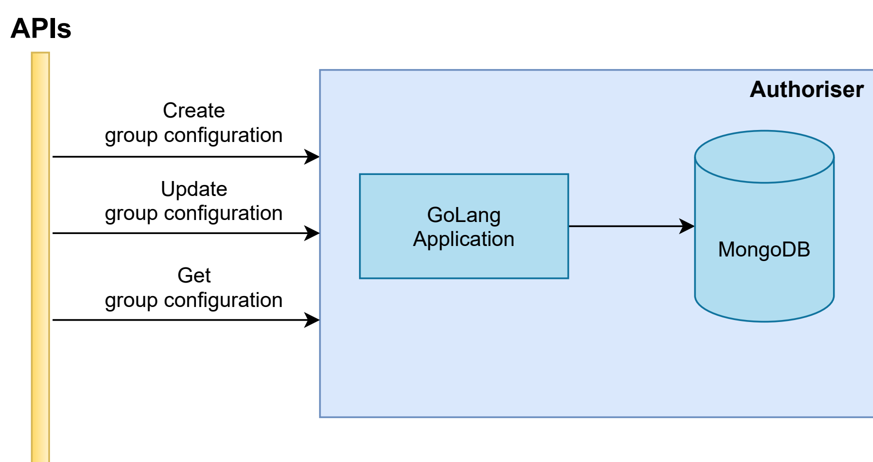


Figure 88: Authoriser: Service Interface

A.1.1 *Functional Requirements*

Listing down the functional requirements for `authoriser`:

- Should allow to create an authorisation configuration per group basis
- Should be able to fetch authorisation configuration per group basis
- Should be able to store the configuration in a loose scheme: fields can be added and removed without much hassle
- Should provide configuration as asked: all data or only the service activated

A.1.2 *Non-Functional Requirements*

Non-functional requirements for this system are as follows:

- Should scale according to the requirements of different APIs:
  - Write API: Create config: less frequent
  - Read API: Fetch Config: highly frequent, called from API Gateway for any request received

- Can afford eventual consistency
- Needs to be highly available, especially for read APIs
- Should push application logs and metrics to appropriate end points

A.2 API GATEWAY

`API-gateway` is in accordance with the architectural patterns recommended for service composition in microservices by Taibi et al. in [107]. Within Branch-Key system, it provides an authentication and authorisation check gateway for file upload and download API calls. It is the only exposed endpoint, in addition to new user registration and login, for the client to access the downstream services. Figure 89 shows an overview of its endpoints and dependencies. Given the nature of its implementation, it is a stateless bottleneck microservice which depends on all other microservices facilitating synchronous communication with the client.

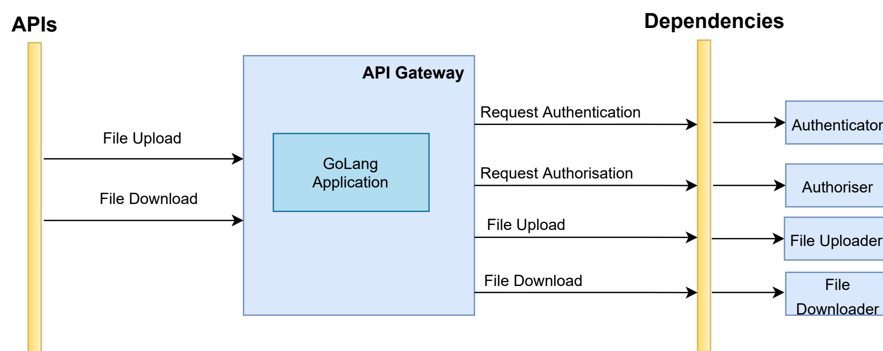


Figure 89: API Gateway: Service Interface

A.2.1 *Functional Requirements*

Following are the functional requirements for `api-gateway`:

- Should provide the "only" exposed endpoint for the client to access the BranchKey system (apart from register and login APIs)
- Should authenticate and authorise every request coming from the client
- Should forward every incoming client request to appropriate downstream service, and relay the response to the client accordingly
- Should rate limit client requests based on IP, group-id, etc.
- Should have a mechanism to push business metrics to a billing service

A.2.2 *Non-Functional Requirements*

The non-functional requirements for `api-gateway` are:

- Should be scaled to handle the client load
- Should have reliable synchronous connectivity with all the needed downstream services
- Should push application logs and metrics to appropriate end points

A.3 AGGREGATION TASK CREATOR

`Aggregation-task-creator` clubs together the required number of user files from the cache, and creates a task for the aggregation. This component works asynchronously, and independent from the client. In coherence with Figure 90, it depends on the event queue for registering the input files uploaded by the clients into the aggregation task, and the corresponding cache to read those files and their corresponding index data.

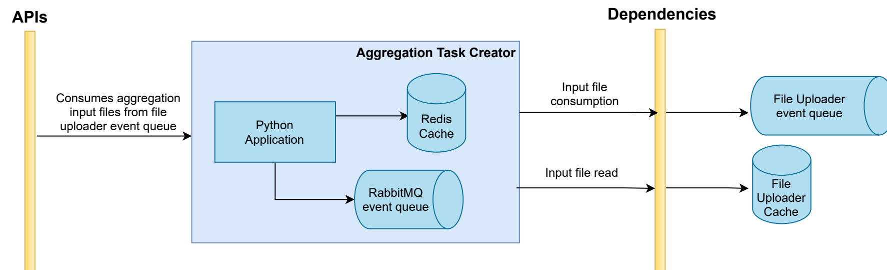


Figure 90: Aggregation Task Creator: Service Interface

A.3.1 *Functional Requirements*

Following are the functional requirements for the `aggregation-task-creator`:

- Should consume the events about incoming files for upload, and make lanes per group-id
- Should create aggregation tasks per lane, with additional information such as group-id and group-config
- Should create segregated lanes so as to allow concurrent aggregation tasks to be processed from a single group-id
- Should create lanes such that input files from one group-id are not inter-mixed with another

A.3.2 *Non-Functional Requirements*

The non-functional requirements for the `aggregation-task-creator` are as follows:

- Should be scaled enough to consume the file upload events within reasonable time
- Should scale smartly, so that once pod consumes all the events from one group-id, or else they might be stuck in deadlocks
- Should be able to access the corresponding cache and event queues within reasonable latencies
- Should push application logs and metrics to appropriate end points

A.4 CENTRAL AGGREGATOR

`Central-aggregator` asynchronously picks up the aggregation tasks from the aggregation task event queue, performs the aggregation, and sends the

aggregated output result file to `file-downloader`, as shown in Figure 91. It depends on the aggregation task queue to read the tasks, the input file cache to read the files, and the File Downloader to send the aggregated output result files.

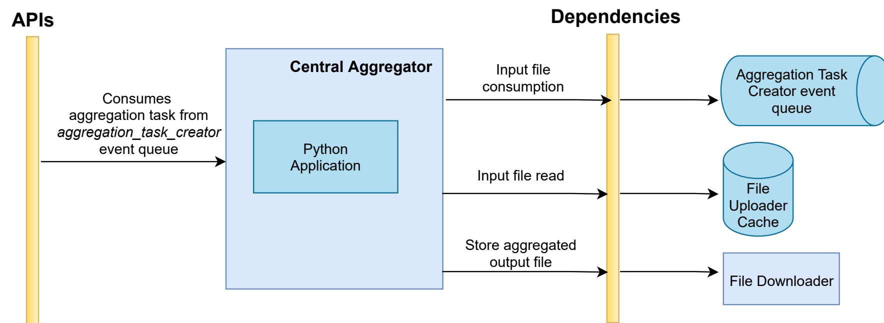


Figure 91: Central Aggregator: Service Interface

A.4.1 *Functional Requirements*

`Central-aggregator` has the following functional requirements:

- Should consume aggregation lane tasks and process them accordingly
- Should send generated output file to the file downloader

A.4.2 *Non-Functional Requirements*

`Central-aggregator` has the following non-functional requirements:

- Should be scaled enough to process the aggregation tasks within reasonable response time
- Should be able to access the corresponding cache and event queues within reasonable latencies
- Should be able to process multiple tasks concurrently
- Should push application logs and metrics to appropriate end points

BIBLIOGRAPHY

[1] *AWS Pricing Calculator*. Available at : https://calculator.aws/#/estimate, accessed on 17 January, 2022.

[2] Daniel Abadi. “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story.” In: *Computer* 45.2 (2012), pp. 37–42. DOI: 10.1109/MC.2012.33.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. “Integrating vertical and horizontal partitioning into automated physical database design.” In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 2004, pp. 359–370. DOI: 10.1145/1007568.1007609.

[4] Amitanand S Aiyer, Eric Anderson, Xiaozhou Li, Mehul A Shah, and Jay J Wylie. “Consistability: Describing Usually Consistent Systems.” In: *HotDep*. 2008.

[5] Kena Alexander, Muhammad Hanif, Choonhwa Lee, Eunsam Kim, and Sumi Helal. “Cost-aware orchestration of applications over heterogeneous clouds.” In: *PloS one* 15.2 (2020), e0228086. DOI: 10.1371/journal.pone.0228086.

[6] Sascha Alpers, Christoph Becker, Andreas Oberweis, and Thomas Schuster. “Microservice based tool support for business process modelling.” In: *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*. IEEE. 2015, pp. 71–78. DOI: 10.1109/EDOCW.2015.32.

[7] *Amazon EKS pricing*. Available at : https://aws.amazon.com/eks/pricing/, accessed on 17 January, 2022.

[8] Vasilios Andrikopoulos, Christoph Fehling, and Frank Leymann. “Designing for CAP-The Effect of Design Decisions on the CAP Properties of Cloud-native Applications.” In: *CLOSER*. 2012, pp. 365–374. DOI: 10.5220/0003931503650374.

[9] Vasilios Andrikopoulos, Steve Strauch, Christoph Fehling, and Frank Leymann. “CAP-oriented design for cloud-native applications.” In: *International Conference on Cloud Computing and Services Science*. Springer. 2012, pp. 215–229. DOI: 10.1007/978-3-319-04519-1_14.

[10] Anthony Anthony and Yaganti Naga Malleswara Rao. “Memcached, Redis, and Aerospike Key-Value Stores Empirical Comparison.” In: ().

[11] Timon Back. “Hybrid serverless and virtual machine deployment model for cost minimization of cloud applications.” PhD thesis. 2018.

[12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. “The potential dangers of causal consistency and an explicit solution.” In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–7. DOI: 10.1145/2391229.2391251.

[13] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. “Bolt-on causal consistency.” In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 761–772. DOI: 10.1145/2463676.2465279.

[14] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services." In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[15] RD Bharati and VZ Attar. "A comprehensive survey on distributed transactions based data partitioning." In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE. 2018, pp. 1–5. DOI: 10.1109/ICCUBEA.2018.8697589.

[16] RD Bharati and VZ Attar. "Workload-Driven Transactional Partitioning for Distributed Databases." In: *Data Intelligence and Cognitive Informatics*. Springer, 2021, pp. 389–396. DOI: 10.1007/978-981-15-8530-2_31.

[17] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, et al. "Towards federated learning at scale: System design." In: *arXiv preprint arXiv:1902.01046* (2019).

[18] *BranchKey*. Available at : https://branchkey.com/, accessed on June 4, 2021.

[19] Susanne Braun, Annette Bieniusa, and Frank Elberzhager. "Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems." In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. 2021, pp. 1–12. DOI: 10.1145/3447865.3457969.

[20] Susanne Braun and Stefan Deßloch. "A Classification of Replicated Data for the Design of Eventually Consistent Domain Models." In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2020, pp. 33–40. DOI: 10.1109/ICSA-C50368.2020.00014.

[21] Susanne Braun, Stefan Deßloch, Eberhard Wolff, Frank Elberzhager, and Andreas Jedlitschka. "Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems: An Action Research Study." In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–11. DOI: 10.1145/3475716.3475771.

[22] Eric Brewer. "Towards robust distributed systems." In: *PODC* (Jan. 2000), p. 7. DOI: 10.1145/343477.343502.

[23] Eric Brewer. "CAP twelve years later: How the "rules" have changed." In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[24] Robson A Campêlo, Marco A Casanova, Dorgival O Guedes, and Alberto HF Laender. "A brief survey on replica consistency in cloud environments." In: *Journal of Internet Services and Applications* 11.1 (2020), pp. 1–13. DOI: 10.1186/s13174-020-0122-y.

[25] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. "Migrating towards microservices: migration and architecture smells." In: *Proceedings of the 2nd International Workshop on Refactoring*. 2018, pp. 1–6. DOI: 10.1145/3242163.3242164.

[26] Tomas Cerny, Michael J Donahoo, and Michal Trnka. "Contextual understanding of microservice architecture: current and future directions." In: *ACM SIGAPP Applied Computing Review* 17.4 (2018), pp. 29–45. DOI: 10.1145/3183628.3183631.

[27] Mohak Chadha, Anshul Jindal, and Michael Gerndt. "Towards Federated Learning using FaaS Fabric." In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 2020, pp. 49–54. DOI: 10.1145/3429880.3430100.

[28] 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." In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26. DOI: 10.1145/1365815.1365816.

[29] Jeang-Kuo Chen and Wei-Zhe Lee. "A study of NoSQL Database for enterprises." In: *2018 International Symposium on Computer, Consumer and Control (IS3C)*. IEEE. 2018, pp. 436–440. DOI: 10.1109/IS3C.2018.00116.

[30] Caio H Costa, PHM Maia, F Carlos, et al. "Sharding by Hash Partitioning." In: *Proceedings of the 17th International Conference on Enterprise Information Systems*. Vol. 1. 2015, pp. 313–320. DOI: 10.5220/0005376203130320.

[31] Balla Wade Diack, Samba Ndiaye, and Yahya Slimani. "CAP theorem between claims and misunderstandings: what is to be sacrificed." In: *International Journal of Advanced Science and Technology* 56 (2013), pp. 1–12.

[32] Miguel Diogo, Bruno Cabral, and Jorge Bernardino. "Consistency models of NoSQL databases." In: *Future Internet* 11.2 (2019), p. 43. DOI: 10.3390/fi11020043.

[33] Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, and Manuel Mazzara. "Microservices: Migration of a mission critical system." In: *arXiv preprint arXiv:1704.04173* (2017). DOI: 10.1109/TSC.2018.2889087.

[34] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: yesterday, today, and tomorrow." In: *Present and ulterior software engineering* (2017), pp. 195–216.

[35] Erki Eessaar. "On pattern-based database design and implementation." In: *2008 Sixth International Conference on Software Engineering Research, Management and Applications*. IEEE. 2008, pp. 235–242. DOI: 10.1109/SERA.2008.24.

[36] Morgan Ekmefjord, Addi Ait-Mlouk, Sadi Alawadi, Mattias Åkesson, Desislava Stoyanova, Ola Spjuth, Salman Toor, and Andreas Hellander. "Scalable federated machine learning with FEDn." In: *arXiv preprint arXiv:2103.00148* (2021).

[37] Mohamed El Kholy and Ahmed El Fatatry. "Framework for interaction between databases and microservice architecture." In: *IT Professional* 21.5 (2019), pp. 57–63. DOI: 10.1109/MITP.2018.2889268.

[38] *Elastic Load Balancing pricing*. Available at : https://aws.amazon.com/elasticloadbalancing/pricing/, accessed on 17 January, 2022.

[39] Weibei Fan, Zhije Han, Yujie Zhang, and Ruchuan Wang. "Method of maintaining data consistency in microservice architecture." In: *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing,(HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE Computer Society. 2018, pp. 47–50. DOI: 10.1109/BDS/HPSC/IDS18.2018.00023.

[40] Alan D Fekete and Krithi Ramamritham. "Consistency models for replicated data." In: *Replication*. Springer, 2010, pp. 1–17. DOI: 10.1007/978-3-642-11294-2_1.

[41] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. "Are architectural smells independent from code smells? An empirical study." In: *Journal of Systems and Software* 154 (2019), pp. 139–156. DOI: 10.1016/j.jss.2019.04.066.

[42] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. "Cluster-based scalable network services." In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. 1997, pp. 78–91.

[43] Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. "Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency." In: *IEEE Software* 35.3 (2017), pp. 63–72. DOI: 10.1109/MS.2017.440134612.

[44] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." In: *SIGACT News* 33.2 (2002), 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[45] Seth Gilbert and Nancy Lynch. "Perspectives on the CAP Theorem." In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389.

[46] Sukhpal Singh Gill and Rajkumar Buyya. "Failure management for reliable cloud computing: A taxonomy, model, and future directions." In: *Computing in Science & Engineering* 22.3 (2018), pp. 52–63. DOI: 10.1109/MCSE.2018.2873866.

[47] *Grafana*. Available at : https://grafana.com/, accessed on 12 January, 2022.

[48] Andreas Grafberger, Mohak Chadha, Anshul Jindal, Jianfeng Gu, and Michael Gerndt. "FedLess: Secure and Scalable Federated Learning Using Serverless Computing." In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 164–173.

[49] J. Gray. "The transaction concept: virtues and limitations." In: 1988.

[50] Ramzi A Haraty and Georges Stephan. "Relational Database Design Patterns." In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. IEEE. 2013, pp. 818–824. DOI: 10.1109/CSE.2013.124.

[51] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. "Decision guidance models for microservices: service discovery and fault tolerance." In: *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*. 2017, pp. 1–10. DOI: 10.1145/3123779.3123804.

[52] Wilhelm Hasselbring and Guido Steinacker. "Microservice architectures for scalability, agility and reliability in e-commerce." In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[53] Joseph M Hellerstein and Peter Alvaro. "Keeping CALM: when distributed consistency is easy." In: *Communications of the ACM* 63.9 (2020), pp. 72–81. DOI: 10.1145/3369736.

[54] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, and Trent Swanson. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft patterns practices, 2014. ISBN: 1621140369. DOI: 10.5555/2636530.

[55] Manar Jammal, Ali Kanso, Parisa Heidari, and Abdallah Shami. "Availability analysis of cloud deployed applications." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2016, pp. 234–235. DOI: 10.1109/IC2E.2016.44.

[56] William Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory." In: *Commun. ACM* 26.2 (1983), 120–125. ISSN: 0001-0782. DOI: 10.1145/358024.358054. URL: https://doi-org.proxy-ub.rug.nl/10.1145/358024.358054.

[57] Samiya Khan, Xiufeng Liu, Syed Arshad Ali, and Mansaf Alam. "Bivariate, Cluster and Suitability Analysis of NoSQL Solutions for Different Application Areas." In: *arXiv preprint arXiv:1911.11181* (2019).

[58] Martin Kleppmann. "A Critique of the CAP Theorem." In: *CoRR* abs/1509.05393 (2015). arXiv: 1509.05393. URL: http://arxiv.org/abs/1509.05393.

[59] Donald Kossmann, Tim Kraska, and Simon Loesing. "An evaluation of alternative architectures for transaction processing in the cloud." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 579–590. DOI: 10.1145/1807167.1807231.

[60] Nicolas Kourtellis, Kleomenis Katevas, and Diego Perino. "Flaas: Federated learning as a service." In: *Proceedings of the 1st Workshop on Distributed Machine Learning*. 2020, pp. 7–13. DOI: 10.1145/3426745.3431337.

[61] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study." In: *Journal of Systems and Software* 126 (2017), pp. 1–16. DOI: 10.1016/j.jss.2017.01.001.

[62] Adam Krechowicz, Stanisław Deniziak, and Grzegorz Łukawski. "Highly Scalable Distributed Architecture for NoSQL Datastore Supporting Strong Consistency." In: *IEEE Access* 9 (2021), pp. 69027–69043. DOI: 10.1109/ACCESS.2021.3077680.

[63] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. "Designing a smart city internet of things platform with microservice architecture." In: *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE. 2015, pp. 25–30. DOI: 10.1109/FiCloud.2015.55.

[64] *Kubegres*. Available at : https://www.kubegres.io/, accessed on 21 December, 2021.

[65] Edward A Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "Quantifying and Generalizing the CAP Theorem." In: *arXiv preprint arXiv:2109.07771* (2021).

[66] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. "Modelling and managing deployment costs of microservice-based cloud applications." In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. 2016, pp. 165–174. DOI: 10.1145/2996890.2996901.

[67] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. "Cloud-Cmp: comparing public cloud providers." In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 1–14. DOI: 10.1145/1879141.1879143.

[68] Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. "Microservices: architecture, container, and challenges." In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2020, pp. 629–635. DOI: 10.1109/QRS-C51114.2020.00107.

[69] Sin Kit Lo, Qinghua Lu, Chen Wang, Hye-Young Paik, and Liming Zhu. "A systematic literature review on federated machine learning: From a software engineering perspective." In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–39.

[70] Sin Kit Lo, Qinghua Lu, Liming Zhu, Hye-young Paik, Xiwei Xu, and Chen Wang. "Architectural patterns for the design of federated learning systems." In: *arXiv preprint arXiv:2101.02373* (2021). DOI: 10.13140/RG.2.2.29934.23365.

[71] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. "Ibm federated learning: an enterprise framework white paper vo. 1." In: *arXiv preprint arXiv:2007.10987* (2020).

[72] Alex Magalhaes, Luciana Rech, Ricardo Moraes, and Francisco Vasques. "REPO: A Microservices Elastic Management System for Cost Reduction in the Cloud." In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2018, pp. 00328–00333. DOI: 10.1109/ISCC.2018.8538453.

[73] Thomas J Marlowe, Cyril S Ku, and James W Benham. "Design patterns for database pedagogy: a proposal." In: *ACM SIGCSE Bulletin* 37.1 (2005), pp. 48–52. DOI: 10.1145/1047124.1047375.

[74] Gastón Márquez and Hernán Astudillo. "Identifying availability tactics to support security architectural design of microservice-based systems." In: *Proceedings of the 13th European Conference on Software Architecture-Volume 2*. 2019, pp. 123–129. DOI: 10.1145/3344948.3344996.

[75] Robert C Martin. "Principles of OOD." In: *URl: http://butunclebob.com/ArticleS. UncleBob. PrinciplesofOod (Last accessed: 2nd July 2015)* (1995).

[76] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Mario Tripiciano, and Alfonso Urso. "The database-is-the-service pattern for microservice architectures." In: *International Conference on Information Technology in Bio-and Medical Informatics*. Springer. 2016, pp. 223–233. DOI: 10.1007/978-3-319-43949-5_18.

[77] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, and Alfonso Urso. "A simplified database pattern for the microservice architecture." In: *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 2016, pp. 35–40. DOI: 10.13140/RG.2.1.3529.3681.

[78] Francesc D Muñoz-Escoí, Rubén de Juan-Marín, José-Ramón García-Escrivá, J R González de Mendívil, and José M Bernabéu-Aubán. “CAP Theorem: Revision of Its Related Consistency Models.” In: *The Computer Journal* 62.6 (2019), pp. 943–960. DOI: 10.1093/comjnl/bxy142.

[79] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. “Vertical Partitioning Algorithms for Database Design.” In: *ACM Trans. Database Syst.* 9.4 (1984), 680–710. ISSN: 0362-5915. DOI: 10.1145/1994.2209. URL: https://doi-org.proxy-ub.rug.nl/10.1145/1994.2209.

[80] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. “Design principles, architectural smells and refactorings for microservices: a multivocal review.” In: *SICS Software-Intensive Cyber-Physical Systems* (2019), pp. 1–13. DOI: [10.1007/s00450-019-00407-8.

[81] Sam Newman. *Building microservices: designing fine-grained systems*. " O’Reilly Media, Inc.", 2015.

[82] *On-Demand Plans for Amazon EC2*. Available at : https://aws.amazon.com/ec2/pricing/on-demand/, accessed on 17 January, 2022.

[83] Felipe Osses, Gastón Márquez, and Hernán Astudillo. “An exploratory study of academic architectural tactics and patterns in microservices: A systematic literature review.” In: *Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018* (2018).

[84] M Tamer Ozsu and Patrick Valduriez. “Distributed database systems: Where are we now?” In: *Computer* 24.8 (1991), pp. 68–78. DOI: 10.1109/2.84879.

[85] Guy Pardon and Cesare Pautasso. “Consistent disaster recovery for microservices: the CAB theorem.” In: *IEEE cloud computing* (2017). DOI: 10.1109/MCC.2018.011791714.

[86] Dan Pritchett. “BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability.” In: *Queue* 6.3 (2008), 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128. URL: https://doi-org.proxy-ub.rug.nl/10.1145/1394127.1394128.

[87] *Prometheus*. Available at : https://prometheus.io/, accessed on 12 January, 2022.

[88] Tilmann Rabl and Hans-Arno Jacobsen. “Query centric partitioning and allocation for partially replicated database systems.” In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 315–330. DOI: 10.1145/3035918.3064052.

[89] *Redis Persistence*. Available at : https://redis.io/topics/persistence, accessed on 3 January, 2022.

[90] *Redis Sentinel*. Available at : https://redis.io/topics/sentinel, accessed on 3 January, 2022.

[91] *Redis latency problems troubleshooting*. Available at : https://redis.io/topics/latency, accessed on 6 January, 2022.

[92] Kim-Thomas Rehmann and Enno Folkerts. “Performance of containerized database management systems.” In: *Proceedings of the Workshop on Testing Database Systems*. 2018, pp. 1–6. DOI: 10.1145/3209950.3209953.

[93] Anja Reuter, Timon Back, and Vasilios Andrikopoulos. "Cost efficiency under mixed serverless and serverful deployments." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 242–245. DOI: 10.1109/SEAA51224.2020.00049.

[94] Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze. "Highly-available applications on unreliable infrastructure: Microservice architectures in practice." In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2017, pp. 130–137. DOI: 10.1109/QRS-C.2017.28.

[95] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering." In: *Empirical software engineering* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[96] Yasushi Saito and Marc Shapiro. "Optimistic replication." In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81. DOI: 10.1145/1057977.1057980.

[97] Tasneem Salah, M Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. "The evolution of distributed systems towards microservices architecture." In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE. 2016, pp. 318–325. DOI: 10.1109/ICITST.2016.7856721.

[98] Santonu Sarkar, Shubha Ramachandran, G Sathish Kumar, Madhu K Iyengar, K Rangarajan, and Saravanan Sivagnanam. "Modularization of a large-scale business application: A case study." In: *IEEE software* 26.2 (2009), pp. 28–35. DOI: 10.1109/MS.2009.42.

[99] Benyamin Shafabakhsh, Robert Lagerström, and Simon Hacks. "Evaluating the Impact of Inter Process Communication in Microservice Architectures." In: *QuASoQ@ APSEC*. 2020, pp. 55–63.

[100] Ali Shakarami, Mostafa Ghobaei-Arani, Ali Shahidinejad, Mohammad Masdari, and Hamid Shakarami. "Data replication schemes in cloud computing: a survey." In: *Cluster Computing* (2021), pp. 1–35. DOI: 10.1007/s10586-021-03283-7.

[101] Salomé Simon. "Brewer's cap theorem." In: *CS341 Distributed Information Systems, University of Basel (HS2012)* (2000).

[102] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. "Analysis of caching and replication strategies for web applications." In: *IEEE Internet Computing* 11.1 (2007), pp. 60–66. DOI: 10.1109/MIC.2007.3.

[103] Michael Stonebraker. "Errors in database systems, eventual consistency, and the cap theorem." In: *Communications of the ACM, BLOG@ ACM* (2010).

[104] Michael Stonebraker. "Errors in database systems, eventual consistency, and the cap theorem." In: *Communications of the ACM, BLOG@ ACM* (2010).

[105] Michael Stonebraker and Rick Cattell. "10 rules for scalable performance in'simple operation'datastores." In: *Communications of the ACM* 54.6 (2011), pp. 72–80. DOI: 10.1145/1953122.1953144.

[106] *Table Partitioning*. Available at : https://www.postgresql.org/docs/13/ddl-partitioning.html, accessed on 6 January, 2022.

[107] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study." In: *CLOSER*. 2018, pp. 221–232. DOI: 10.5220/0006798302210232.

[108] *Taints and Tolerations*. Available at : https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/, accessed on 12 January, 2022.

[109] *The Go Memory Model*. Available at : https://go.dev/ref/mem#tmp_5, accessed on 12 January, 2022.

[110] Rafik Tighilt, Manel Abdellatif, Nader Abu Saad, Naouel Moha, and Yann-Gaël Guéhéneuc. "Collection and Identification Of Microservices Patterns And Antipatterns." In: July 2021.

[111] José A Valdivia, Xavier Limón, and Karen Cortes-Verdin. "Quality attributes in patterns related to microservice architecture: A Systematic Literature Review." In: *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE. 2019, pp. 181–190. DOI: 10.1109/CONISOFT.2019.00034.

[112] Harley Vera-Olivera, Ruizhe Guo, Ruben Cruz Huacarpuma, Ana Paula Bernardi Da Silva, Ari Melo Mariano, and Maristela Holanda. "Data Modeling and NoSQL Databases-A Systematic Mapping Review." In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–26. DOI: 10.1145/3457608.

[113] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. "Synapse: a microservices architecture for heterogeneous-database web applications." In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–16. DOI: 10.1145/2741948.2741975.

[114] Patrick Vogel. "Computing the Cost and Waste in Cloud Computing Monitoring." 2019.

[115] Werner Vogels. "Eventually consistent." In: *Communications of the ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432.

[116] Hiroshi Wada, Alan D Fekete, Liang Zhao, Kevin Lee, and Anna Liu. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective." In: *CIDR*. Vol. 11. 2011, pp. 134–143.

[117] Stefan Walraven, Eddy Truyen, and Wouter Joosen. "Comparing PaaS offerings in light of SaaS development." In: *Computing* 96.8 (2014), pp. 669–724. DOI: 10.1007/s00607-013-0346-9.

[118] Hao Wang, Di Niu, and Baochun Li. "Distributed machine learning with a serverless architecture." In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1288–1296. DOI: 10.1109/INFOCOM.2019.8737391.

[119] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. "CloudTPS: Scalable transactions for Web applications in the cloud." In: *IEEE Transactions on Services Computing* 5.4 (2011), pp. 525–539. DOI: 10.1109/TSC.2011.18.

[120] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. "Microservice architecture in reality: An industrial inquiry." In: *2019 IEEE international conference on software architecture (ICSA)*. IEEE. 2019, pp. 51–60. DOI: 10.1109/ICSA.2019.00014.

[121] Xi Zheng. “Database as a service-current issues and its future.” In: *arXiv preprint arXiv:1804.00465* (2018).

[122] Weiming Zhuang, Xin Gan, Yonggang Wen, and Shuai Zhang. “EasyFL: A Low-code Federated Learning Platform For Dummies.” In: *arXiv preprint arXiv:2105.07603* (2021). DOI: 10.1109/JIOT.2022.3143842.

[123] O Zimmermann. “Mircroservices tenets: Agile approach to service development and deployment.” In: *Proceedings of the Symposium/Summer School on Service-Oriented Computing*. 2016. DOI: 10.1007/s00450-016-0337-0.