INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Comparing JSON and Protocol Buffers in HTTP-based REST architectures: performance and energy efficiency

MIGUEL ALVES FERREIRA
junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Comparing JSON and Protocol Buffers in HTTP-based REST architectures: performance and energy efficiency

Miguel Alves Ferreira

Dissertation

Master in Informatics Engineering

Specialisation in Software Engineering

Supervisor: Isabel Azevedo

Porto, 06 2025

Integrity Statement

I declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section “Ethical considerations” of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, 25 de June de 2025

Summary

Nos últimos anos, tem-se verificado um interesse crescente em melhorar o desempenho dos sistemas de software. Ao mesmo tempo, a eficiência energética surgiu como uma preocupação significativa nesta nova década [1]. Isto deve-se principalmente a dois problemas crescentes: a correlação entre um maior consumo de energia e o aumento dos custos [2], e as crescentes preocupações ambientais [3]. Consequentemente, foi desenvolvida uma grande variedade de boas práticas e padrões[2] para sistemas de software.

Além disso, apesar do vasto leque de tecnologias disponíveis, não se verifica uma utilização predominante destas tecnologias em conjunto. Apesar do potencial de melhoria do desempenho e da eficiência energética que tais combinações podem oferecer.

Um exemplo disso é a integração dos Protocol Buffers com a arquitetura HTTP REST. Já está estabelecido que os Protocol Buffers são uma solução comprovada para melhorar o desempenho [4]. Contudo, não existe uma comparação dos Protocol Buffers neste tipo de arquitetura com outros formatos de serialização estabelecidos, como o JSON.

Este estudo foi assim concebido para comparar os Protocol Buffers com o JSON numa arquitetura HTTP REST em termos de desempenho e de eficiência energética, de forma a verificar a viabilidade dos Protocol Buffers neste ecossistema. Para atingir este objetivo, foi necessário migrar um projeto Java open source que possuía um dos mecanismos de serialização para o seu homólogo.

Os resultados indicam que, para métricas de desempenho como o rendimento e o tempo médio de resposta, a utilização de Protocol Buffers pode ser mais vantajosa quando o endpoint em avaliação exige uma serialização substancial. Além do mais, no que respeita à eficiência energética, o mesmo padrão é observado, ou seja, nos casos em que a serialização é mais intensa, os Protocol Buffers tem melhor efficiência energética.

Palavras-chave: JSON, Protocol Buffers, REST, serialização, desempenho, eficiência energética

Abstract

In recent years, there has been an increasing interest in enhancing the performance of software systems. Concurrently, energy efficiency has emerged as a significant concern in this new decade [1]. This is primarily due to two factors: the correlation between higher energy consumption and increased costs [2], and the growing environmental concerns [3]. Consequently, a wide variety of good practices and patterns [2] for software systems have been developed.

Furthermore, despite the extensive array of technologies available, there is an absence of prevalent utilisation of these technologies in conjunction with one another. This is despite the potential for enhanced performance and energy efficiency that such combinations could offer.

One such example is the integration of Protocol Buffers with the HTTP REST architecture. It is already established that Protocol Buffers are a proven solution for enhanced performance [4]. Consequently, there is a lack of comparison of Protocol Buffers in this type of architecture with other established serialisation formats, such as JSON.

The present study was thus conceived to compare Protocol Buffers to JSON in an HTTP REST architecture in terms of performance and energy efficiency, to ascertain the viability of Protocol Buffers in this ecosystem in terms of performance and energy efficiency. In order to achieve this objective, it was necessary to migrate an open-source Java project that possessed either one of the serialisation mechanisms to its counterpart.

The findings indicate that, for performance metrics such as throughput and average response time, Protocol Buffers demonstrate superiority when the endpoint under evaluation requires substantial serialisation. Furthermore, with regard to energy efficiency, a similar trend is evident in cases where serialisation is more intense. In such cases, Protocol Buffers demonstrate superior energy efficiency.

Keywords: JSON, Protocol Buffers, REST, serialisation, performance, energy efficiency

Acknowledgements

During secondary school, I never thought I could achieve higher education, let alone a master's degree. I was a lazy student and had no vision for my future. That reality changed because of a biology teacher called Marcia Pacheco, who never gave up on any student, including me, and, for that, I am deeply grateful to her for everything she did for me.

In my second semester of my degree, I had the special pleasure of being a student of Professor Paula Morais, who managed to revive my interest in technology. She is also the main reason I know how to program, and for that, I am incredibly grateful. I would also like to thank Professor Fátima Lopes for motivating me to pursue a master's degree, something I didn't even consider doing during my degree and wouldn't have done if it hadn't been for her.

I would like to express my gratitude to my supervisor for the guidance and support provided throughout the dissertation development process.

Finally, and most importantly, I would like to thank my family from the bottom of my heart for their unconditional support.

Without these incredible people, I wouldn't be the person I am today, and I wouldn't be where I am.

Index

List of Figures

List of Tables

List of Code Snippets

List of abbreviations

API	*Application Programming Interface*
CPU	*Central Processing Unit*
CRUD	*Create, Read, Update, Delete*
DTO	*Data Transfer Object*
EBPF	*Extended Berkeley Packet Filter*
GRPC	*Google Remote Procedure Call*
GDPR	*General Data Protection Regulation*
Go	*Golang*
GQM	*Goal Question Metric*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IDL	*Interface Definition Language*
IoT	*Internet Of Things*
JAR	*Java Archive*
JPA	*Java Persistence API*
JSON	*JavaScript Object Notation*
JVM	*Java Virtual Machine*
JWT	*JSON Web Token*
Kepler	*Kubernetes Efficient Power Level Exporter*
MIT	*Massachusetts Institute of Technology*
Mb	*Megabytes*
POJO	*Plain Java Old Object*
REST	*Representational State Transfer*

SOAP *Simple Object Access Protocol*

URI *Uniform Resource Identifier*

Vu *Virtual User*

WBS *Work Breakdown Structure*

XML *Extensible Markup Language*

YAML *YAML Ain't Markup Language*

1 Introduction

This is the opening chapter of the dissertation for the Master in Informatics Engineering at Instituto Superior de Engenharia do Porto. The structure of this chapter is divided into six sections. The initial section is the context section, which provides an overview of the current state of certain technologies related to the work.

The subsequent section delineates the problem description, which elucidates the issues present with current technologies and the state of development practices related to the theme. The following section is dedicated to the objectives and approach, which describe the problem, the research questions developed, and the research methodology.

The final two sections address ethical considerations and document structure. The former details the ethical principles that were adhered to throughout the project, while the latter describes the present document's structure.

1.1 Context

REST (Representational State Transfer) Application Programming Interfaces (APIs) rely heavily on serialisation formats like JavaScript Object Notation (JSON), Extensible Markup Language (XML), Protocol Buffers and others. JSON has become a widely adopted format due to its simplicity, human readability, and wide support across programming languages. However, JSON's text-based, schema-less nature results in larger payloads, slower parsing times, and increased network overhead, especially for large datasets [4], [5].

Nevertheless, a multitude of alternative serialisation formats could be used to mitigate the undesirable consequences of using JSON. Some examples of such formats include Apache Avro [6], Message Pack [7], Thrift [8] or the focus of this study, the Protocol Buffers [9]. These previously referred serialisation formats are classified as binary serialisation and can address some of the issues encountered with JSON.

Moreover, Protocol Buffers [10] were developed by Google and offer a more efficient, compact, and schema-driven alternative to JSON. Its binary encoding reduces payload sizes and improves parsing speed, making it ideal for high-performance applications with stringent resource constraints [11], [12], [13]. Despite these advantages, Protocol Buffers' complexity and lack of human readability might limit its adoption in REST architectures, where JSON remains the dominant choice [14].

1.2 Problem Description

Choosing the right technology for data serialisation plays a key role in the performance and scalability of REST applications, as it is integrated from the start to the finish of the project [15]. JSON is the most popular serialisation format used because of its simplicity, human readability, and support across programming languages, but it has drawbacks that could be improved by instead using Protocol Buffers in the same Hypertext Transfer Protocol (HTTP) REST application.

Normally, Protocol Buffers are used with Google Remote Procedure Call (gRPC). Which typically uses Protocol Buffers over HTTP/2 and is often observed to have performance advantages over traditional RESTful APIs implemented over HTTP/1.1 [16], [17].

Moreover, performance and energy efficiency are increasingly critical in today's software development, particularly in HTTP-based REST architecture, which is correlated to green computing, also called sustainable computing [18], which is about making proper design decisions in the life cycle of the system so that it has a reduced carbon footprint.

Consequently, increased latency, reduced throughput, or inefficient resource usage can significantly impact user experience and system scalability. On top of that, research has shown that underutilised or poorly optimised systems lead to higher costs, resource wastage, and scalability issues [19], [20]. For those reasons, serialisation formats like Protocol Buffers could reduce these bottlenecks by minimising payload sizes and improving processing speed.

Furthermore, energy efficiency is another growing concern, especially in environments like mobile devices, IoT, and data centres. With that being said, researchers have shown that energy consumption impacts operational costs and plays an important role in system design decisions and the sustainability of large-scale deployments [20], [21], for example, in modern data centres, serialisation optimisations can lead to substantial energy savings, translating into lower costs and reduced environmental impact [2], [20].

Moreover, the field of energy efficiency is growing so much that a foundation [22] was created. The foundation raises awareness of the importance of energy-efficient software and provides guidance on green software practices and patterns [2] to improve system energy efficiency.

1.3 Objectives and Approach

The primary objective of the present study is to conduct an experiment to assess the performance and energy efficiency of using Protocol Buffers in the same HTTP-based REST architecture.

Given the commonality of JSON as the prevailing serialisation format for such architectures, it has been designated as the control variable. This designation establishes the baseline against which the experiments are conducted.

To perform this investigation, certain components of a project that utilises JSON as the primary serialisation format have been migrated to Protocol Buffers. This migration enables a comparative analysis, thus giving rise to the formulation of the following research questions:

- RQ1 How does using Protocol Buffers impact the performance of HTTP-based REST architectures compared to JSON?
- RQ2 To what extent can Protocol Buffers improve energy efficiency in HTTP-based REST architecture communications compared to JSON?

Thus, the research methodology chosen for this study is based on controlled experiment research [23]. The primary goal is to evaluate the impact of using Protocol Buffers compared to JSON as a serialisation format in REST applications. This study aims to address a significant gap in existing literature, where limited empirical evidence is available on the application of Protocol Buffers within the REST architectural style, despite their known advantages in other contexts such as gRPC.

A controlled experiment is the most suitable methodology for this study because it allows precise evaluation of causal relationships by isolating the effect of the serialisation format (Protocol Buffers vs. JSON) on performance metrics such as speed and resource utilisation [23], [24]. They ensure consistency and repeatability by applying the same conditions across tests, minimising confounding factors. This methodology emphasises internal validity, ensuring that observed differences are due to the serialisation method rather than external influences like hardware or network variability [25].

To ensure accuracy and credibility, the research incorporates statistical analysis methods to interpret results. By addressing inefficiencies in JSON and exploring the applicability of Protocol Buffers in REST APIs, the study aims to deliver relevant and impactful insights capable of informing future decisions in API design and optimisation.

Furthermore, to apply this research methodology, a project was found with a predefined criterion outlined in the Experiment chapter, which ensured the absence of any bias toward the project under consideration. Also, the selected project was analysed from a performance and energy consumption perspective in the Analysis chapter, from which the data was extracted and statistically analysed to conclude the work done.

1.4 Ethical Considerations

This study adheres to ethical principles to ensure integrity and transparency. Although the research does not involve human subjects or sensitive data related to any subjects, the study makes sure all test data used is synthetic and generated specifically for benchmarking purposes, ensuring compliance with the General Data Protection Regulation (GDPR).

All setup, tools, and methodologies are publicly documented in a repository. To ensure the integrity of performance evaluations, the project is completely devoid of any form of bias. To ensure fair comparisons, both serialisation formats are evaluated under equal conditions whenever possible.

All source code developed for this study is publicly available from its inception to its conclusion, following the principles established from the Open Science principles [14]. Furthermore, a Massachusetts Institute of Technology (MIT) license is issued for the free use of any content produced by the study, allowing for further scrutiny of the results obtained, usage for additional studies, or further improvement of the study.

In conclusion, the code of ethics established by the Institute of Electrical and Electronics Engineers (IEEE) [26] was followed with complete adherence to consistently achieve the highest standards and act in the best interests of the public. In addition, the code of ethics and deontology of the Order of Engineers was also followed with the utmost diligence [27].

1.5 Document Structure

The dissertation is comprised of a total of six chapters. The initial chapter is the Introduction, which presents the theme and objective of the project under development. It also delineates the problem to be addressed, the research questions to be investigated, the research methodology, and the ethical considerations that were taken into account in the development and procedures.

The next chapter, the Background, provides a detailed overview of the technologies observed in the context of this research, including JSON and Protocol Buffers. Additionally, it addresses the auxiliary technologies that are used in the research process.

Moreover, the Literature Review chapter is intended to address the research question and all related aspects of the research process, including the methods and rationale behind data collection from the literature and the selection of salient data for further review in the dissertation.

The Planning chapter delineates the management of the project, including the identification of stakeholders, the delineation of the scope, and the articulation of objectives. This section also demonstrates the work to be done and the timeline associated with that work. To finalise the

planning chapter, it also exhibits the skills required for the project to be successful, the strengths and weaknesses of the author, and the plans to improve these skills.

The subsequent chapter, namely, the Experiment, addresses the project selection for the dissertation, its architecture, the applied alterations, the implementation details of the migration, the experiment performed on the project with data collection and a discussion on the collected data.

Finally, the final chapter of the dissertation offers a comprehensive overview of its accomplishments, the challenges encountered during its development, and the potential threats to its validity.

Additionally, the references and all the appendices that assist the study in the dissertation are located at the end of the document.

2 Background

The subsequent chapter aims to provide information on fundamental concepts related to the work that was developed. In accordance with this objective, each primary section of the present chapter focuses on either a technology or a concept related to the work, including the fundamentals of REST, JSON, Protocol Buffers, and performance and energy analysis tools.

2.1 REST

The REST architectural style was introduced in the year 2000 [28]. It defines a set of standards and constraints used to build scalable systems. Due to its simplicity and adherence to standard protocols such as HTTP and Hypertext Transfer Protocol Secure (HTTPS), it facilitates its integration into web browsers, mobile apps, or even command-line tools like curl. It provides a lightweight stateless client-server communication model, which enables applications to be scalable.

Consequently, REST has established itself as a significant component of current architectural frameworks, such as microservices and cloud computing [29].

At its core, REST leverages the HTTP methods like GET, POST, PUT, DELETE, and PATCH to perform operations on resources. These methods correspond to common create, read, update, delete (CRUD) operations.

- GET: Retrieves a resource or collection of resources.
- POST: Creates a new resource.
- PUT: Updates an existing resource or creates it if it doesn’t exist.
- DELETE: Removes a resource.
- PATCH: Partially updates a resource.

Furthermore, REST has a flexible design, allowing for different serialisation formats such as JSON, XML, or Protocol Buffers. Not only that, but it is also highly scalable and has overall reduced costs, making it a common choice in the development of systems.

However, REST has been criticised in certain respects, for example, it has been shown to have high latency, specifically in the case of multiple client-server interactions, which can result in increased response times. Another issue that has been identified is the high bandwidth usage that occurs when certain verbose formats, such as JSON or XML, are employed, which could lead to an increase in energy consumption.

2.1.1 How are REST Applications Structured

REST applications follow a well-defined structure and set of principles to ensure scalability, maintainability, and interoperability. These principles are defined as REST constraints, which describe how REST APIS should behave [28], [30].

- Statelessness: Every client-server interaction must be independent. The server does not store any information about previous requests; instead, each request must contain all the necessary information to process it. This ensures scalability and simplifies server implementation.
- Client-Server Architecture: This states that the client and server are separate entities, the client is responsible for the user interface and request initiation, while the server manages data storage and business logic.
- Uniform Interface: REST applications must follow a standardised way of interacting with resources. This includes:
    - Resource Identification, which is done through Uniform Resource Identifiers (URIs), like the following: /users/1 identifies a specific user resource.
    - Standardised methods from the HTTP, such as GET, POST, PUT, and DELETE, facilitate the execution of operations on resources.
    - Representation of resources is represented in formats like JSON, XML, or Protocol Buffers.
    - Hypermedia as The Engine of Application State (HATEOAS) is the principle that clients should navigate the application through hyperlinks embedded in responses. This allows for the dynamic discovery of resources and actions.
- Layered System: REST applications can be designed with multiple layers (e.g., caching servers, authentication layers, load balancers) that operate independently. The client interacts with the server as if it were a single entity, without being aware of intermediate layers.
- Cacheability: The server should indicate if the responses are cacheable, which results in reduced latency and bandwidth usage.
- Code on Demand: It allows for the server to send executable code to the client.

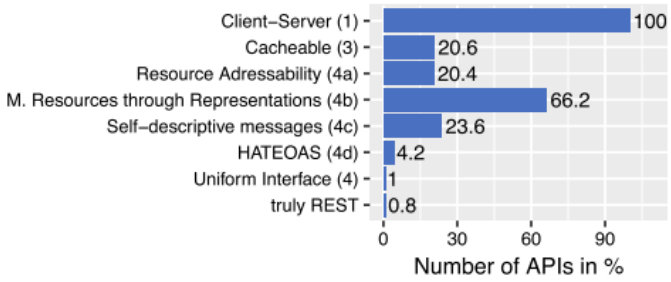

Figure 1 - Overall compliance with the REST principles of 500 REST APIS
Reprinted from [14]

Although REST has become the dominant architectural style for web services, studies show that many implementations deviate significantly from its core principles, as seen in Figure 1. For instance, a large-scale analysis of 500 public REST APIs revealed that only 0.8% of the services fully conformed to all REST architectural principles [14].

2.2 JSON

JSON was introduced in 2001 and has become one of the most widely used data serialisation formats in web and application development. It was created to provide a lightweight, easy-to-parse format for transmitting structured data. In addition, its philosophy of efficiency and simplicity has allowed it to become an integral foundation of modern software systems, which allow data to be exchanged in real time with minimal overhead [31].

At its core, JSON represents data as key-value pairs. The keys are always strings, while values can be of various data types, including strings, numbers, booleans, arrays, objects, or null. JSON's syntax is intuitive and precise, and it adheres to the following rules.

- Keys must be enclosed in double quotes (").
- Objects are encapsulated within curly braces ({}) and consist of key-value pairs separated by colons (:).
- Arrays are enclosed in square brackets ([]) and represent ordered lists of values.
- Values can include primitive data types (e.g., strings or numbers) or complex structures like nested objects and arrays.

Code Snippet 1 is an example of a JSON object that demonstrates previously described principles.

```
{
    "name": "John Doe",
    "age": 30,
    "address": {
    "street": "123 Main St",
        "city": "Anytown",
    },
    "phone": [
     "123-456-7890",
     "987-654-3210"
    ]
}
```

Code Snippet 1 - Example of a user JSON

In this example, the name and age fields represent simple string and numeric values, respectively, while the address field is a nested object containing its key-value pairs. The phone value demonstrates the use of arrays to store multiple items. Furthermore, the flexibility of JSON also allows for the creation of complex data structures, which makes it great for most applications.

Additionally, JSON's popularity stems from its simplicity, versatility, and widespread use in various environments. Its extensive use in APIs enables seamless data exchange between servers and clients, and it is also present in configuration files for storing application settings and in real-time communication systems, such as chat applications and IoT devices, where lightweight and efficient data representation is essential [4], [11], [13].

Another key feature of JSON is its ease of parsing and generation, which is demonstrated by the fact that most modern programming languages provide built-in libraries or modules to manage it, thus allowing developers to effortlessly serialise and deserialise data. For example, in Golang (Go), JSON encoding and decoding are facilitated by the encoding/json package.

Code Snippet 2 shows the encoding and decoding of JSON data can be achieved in Go with the previously defined JSON object.

```
package main
import (
"encoding/json"
"fmt"
)
type Person Struct {
    Name string `json: "name"`,
    age int `json: "age"`,
    Address struct{
        Street string `json:"street"`,
        City string `json:"city"`,
    }
    Phone []string `json:"phone"`,
}

func main() {
    jsonData := `{
        "name": "John Doe",
        "age": 30,
        "address": {
            "street": "123 Main St",
            "city": "Anytown",
        },
        "phone": ["123-456-7890", "987-654-3210"]
    }`
    var person Person
    err := json.Unmarshal([]byte(jsonData), &person) // Decoding JSON data
into Go struct
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Decoded Struct: %+v\n", person)

    encodedData, err := json.Marshal(person) // Encoding Go struct into
JSON data
    if err != nil {
        fmt.Println("Error encoding JSON:", err)
        return
    }
    fmt.Println("encode JSON: ", string(encodedData))
}
```

Code Snippet 2 - Example of decoding and encoding a user JSON in Go

In Code Snippet 2, the person struct defines the expected structure of the JSON object. The json.Unmarshal function is used to parse JSON into this struct, while json.Marshal converts the struct back into a JSON byte slice.

2.3 Protocol Buffers

Protocol Buffers, commonly referred to as Protobufs, were initially developed internally at Google in 2001 and released to the public in 2008. According to the official documentation,

"Protocol buffers are language-neutral, platform-neutral extensible mechanisms for serialising structured data" [9].

Protocol Buffers are fundamentally a binary serialisation format that facilitates efficient and compact data exchange between applications. Its design prioritises performance, compactness, and simplicity, making it an ideal choice for scenarios where high throughput and low latency are critical [13], [32], [33], [34].

Protocol Buffers are extensively used in applications such as microservices, distributed systems, and IoT devices, where efficiency in data communication is paramount. A prominent use case is its integration with gRPC, which is a high-performance framework for inter-service communication which leverages Protocol Buffers for serialising and deserialising messages. This combination enables developers to build scalable and efficient APIs and distributed systems [16], [35].

However, while Protocol Buffers are often noted for outperforming other serialisation formats like XML and JSON in terms of speed and data compactness, comparisons must consider architectural differences. For instance, evaluations frequently compare Protocol Buffers in gRPC-based architectures against JSON in REST architectures [12], [35], [36].

Such comparisons may introduce variability in the results of certain metrics due to the intrinsic differences between the two paradigms, such as transport protocol overhead and communication patterns.

2.3.1 Structure of Protocol Buffers

Protocol Buffers use a schema-based serialisation approach, meaning the data structure must be explicitly defined in a schema file with the .proto extension. This schema, written in Protocol Buffers Interface Definition Language (IDL), outlines the data structure, specifying field names, types, and nested structures. The schema also includes metadata, such as the syntax version, which defaults to Proto2 unless explicitly specified as Proto3. Code Snippet 3 is a .proto file that defines message types for a person and their address.

```
syntax = "proto3"; // Here is the version of the syntax if omitted proto2
is assumed
message Person {
    string name = 1;
    int32 id = 2;
    Address address = 3;
    repeated string phone = 4;
}

message Address {
    string street = 1;
    string city = 2;
}
```

Code Snippet 3 - Block of code from a .proto file defining a person

In Code Snippet 3 it is possible to see the following rules.

- Each Field is assigned a unique number, used as a field identifier in the binary encoding.
- The repeated keyword indicates that a field can hold multiple values, effectively defining a list or array.
- Nested message types, such as address within person, allow for the definition of hierarchical data structures.

Protocol Buffers support a rich set of data types to address diverse use cases, Table 1 displays the different data types supported by Protocol Buffers.

Table 1 - List of Types that can be defined in the .proto file

Type	**Description**
double	Double Precision floating point
float	Single precision floating point
int32	Signed 32-bit integer
int64	Signed 64-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer
sint32	Signed variable-length integer
sint64	Signed variable-length integer
bool	Boolean value
string	String
bytes	Bytes
fixed32	32-bit fixed-point
fixed64	64-bit fixed point
sfixed32	32-bit signed fixed-point
sfixed64	64-bit signed fixed-point

2.3.2 Schema Compilation and Code Generation

Once the .proto schema file is created, it must be compiled using the protoc compiler, which generates language-specific code based on the schema definition. This generated code provides data structures and methods for encoding and decoding Protocol Buffers messages. For example, in Go, the protoc compiler produces Go structs and associated methods to interact with the data, these files typically have a .pb.go extension in Go projects.

2.4 Performance and Energy Analysis Tools

The utilisation of performance and energy analysis tools is imperative for the evaluation of software systems in terms of efficiency and resource utilisation. These tools enable the observation of applications under diverse workloads and operational conditions. This section introduces the concepts of performance testing and energy consumption analysis, followed by examples of commonly used tools.

2.4.1 Performance Testing

Performance testing is defined as the process of assessing a system's behaviour under a specific workload. This assessment involves the evaluation of time behaviours, resource utilisation, and capacity [37]. Furthermore, a wide variety of performance tests exists, each designed to fulfil a distinct purpose.

Consequently, some tests are more common than others. One such example of the most common tests is the load test, which analyses the system's capacity to manage variable expected loads, enabling to identification of bottlenecks in software systems. An additional test is the stress test, which analyses a system's performance under maximum loads, revealing its vulnerabilities and points of failure [37], [38].

Moreover, an additional frequently employed performance test is the endurance test, which subjects a system to a sustained, prolonged load, exposing memory leaks and resource exhaustion. The last example of a performance test is the spike test, which simulates sudden, extreme loads to expose weaknesses and vulnerabilities [37], [38].

Another and final example of a performance test is the spike test, which subjects the system to a sudden extreme load, simulating scenarios where there is an unexpected increase in system traffic, exposing the potential weaknesses and vulnerabilities of the system [37], [38].

A variety of tools exist to perform performance testing. One of the tools under consideration is JMeter [39], a Java GUI-based testing tool capable of a wide variety of tests due to its plugin support. In addition to its other advantages, JMeter offers out-of-the-box reporting capabilities and supports data-driven testing using CSV files, allowing for easy parameterisation.

Another tool is Grafana k6 [40], which allows writing scripts in JavaScript for performance testing, also features a range of plugins and enables the integration of other NPM packages via Webpack.

2.4.2 Energy consumption analysis

Energy consumption analysis is defined as the process of measuring, modelling, and evaluating the power usage of software systems. identification of opportunities for enhancement of energy efficiency, performance, operational cost reduction, and environmental sustainability [41].

Some tools have been developed to perform the energy consumption analysis. One such tool is PowerAPI [42], an open-source tool capable of delivering real-time estimations of software consumption through various sensors, including physical meters, processor interfaces, hardware counters, and OS counters. This process is facilitated by the PowerMeter software application, which is built with PowerAPI. The application can be divided into two components, the sensors component and the formula component.

The second one is Kubernetes Efficient Power Level Exporter (Kepler) [43], [44], which is an open-source tool designed to monitor and estimate the energy consumption of Kubernetes workloads. By leveraging technologies like extended Berkeley Packet Filter (EBPF) and machine learning models, Kepler provides detailed insights into power usage at the process, container, and pod levels. It also provides quite a few features like. The EBPF and the hardware counters enable the collection of performance metrics directly from the Linux kernel. Moreover, the use of hardware counters facilitates the acquisition of detailed information regarding energy consumption estimates.

3 Literature Review

This section is about analysing research that analyses the performance of JSON and Protocol Buffers, and the possible impacts of using Protocol Buffers in REST applications. This chapter outlines the data sources, keywords, and inclusion and exclusion criteria used for the research process, with the main objective of the chapter being to respond to the research questions and objectives of the study.

The present chapter is comprised of a total of seven main sections. The initial section presents the research questions that were previously outlined. The data sources section, the search terms section, the eligibility criteria section and the data collection process delineate the process that defines the information sources, the information retrieved, and the utilisation of information for the literature review.

Finally, the discussion and conclusion sections describe the findings of the literature review that are suitable for the research at hand.

3.1 Research Questions

As stated in section 1.3 the research questions are.

RQ1. How does using Protocol Buffers impact the performance of HTTP-based REST architectures compared to JSON?

RQ2. To what extent can Protocol Buffers improve energy efficiency in HTTP-based REST architecture communications compared to JSON?

3.2 Data Sources

Data sources are crucial in the research process, as they provide indexed literature that can be used to answer research questions and objectives.

Table 2 - Data sources

Identifier	Database	URL
DS1	Google	https://scholar.google.com/
DS2	ACM Digital Library	https://dl.acm.org/
DS3	B-ON	https://www.b-on.pt/
DS4	IEEE Xplore	https://ieeexplore.ieee.org/Xplore/home.jsp

These digital libraries have many indexed sources of data, like articles, papers, and books, which are reviewed by experts in the field.

3.3 Search Terms

The subsequent section delineates the most important keywords for the problem previously identified in the Section 1.2. The keywords identified as relevant include Protocol Buffers, Protobufs, JSON, REST, Performance, Serialisation, Deserialization, Efficiency, latency, Resource consumption, and Energy consumption.

A search query was formulated using these keywords to identify relevant studies from the data sources outlined in Table 2. The query is provided in the following code snippet.

```
("Protocol Buffers" OR "Protobufs")
AND ("HTTP REST" OR "HTTP-based REST" OR "REST API")
AND "JSON" AND ("serialization" OR "deserialization")
AND ("performance" OR "latency" OR "resource consumption" OR "efficiency"
OR "energy consumption")
-security
```

Code Snippet 4 - Search query developed with the search terms and research questions

Code Snippet 4 was developed to systematically explore literature related to the use of Protocol Buffers and REST in the context of serialisation and deserialisation processes. The query incorporates key terms such as "JSON" and "XML" to compare serialisation frameworks commonly used in web applications, mobile platforms, IoT, and microservices. Furthermore, performance, latency, efficiency, resource consumption, and energy consumption were included to focus on studies evaluating system optimisation.

3.4 Eligibility Criteria

The inclusion criteria for the literature review are as follows.

- IC1: Studies examining Protocol Buffers, JSON, as data serialisation.
- IC2: Research measuring energy consumption, central processing unit (CPU) power usage, battery consumption, or memory usage associated with REST API communication.
- IC3: Studies providing serialisation, deserialization, transmission efficiency, or resource usage metrics related to energy consumption.
- IC4: Studies covering programming languages and platforms relevant to REST APIs

The exclusion criteria are as follows.

- EC1: Studies focusing on unrelated aspects such as security, data integrity, or accuracy without addressing energy consumption or efficiency.
- EC2: Studies in non-REST API environments or using protocols such as gRPC or simple Object Access Protocol (SOAP), unless specifically measuring Protocol Buffers.
- EC3: Studies using outdated versions of Protocol Buffers or JSON libraries that are no longer relevant to current REST API technology.

3.5 Data Collection Process

The Prisma systematic methodology [45] was employed to guide the literature review process. This methodology involves three steps.

- Identification: Searching for relevant studies in digital libraries using the search query.
- Screening: All retrieved articles were evaluated, then after an analysis, if they were relevant to the research topic and research questions, they were included in the review.
- Inclusion: All the studies that were relevant to the research questions and objectives were included in the review process.

This is a key step of the research as it can provide valuable insight into the quality of the research and the relevance of the data to the research questions. Figure 2 shows the Prisma flowchart.

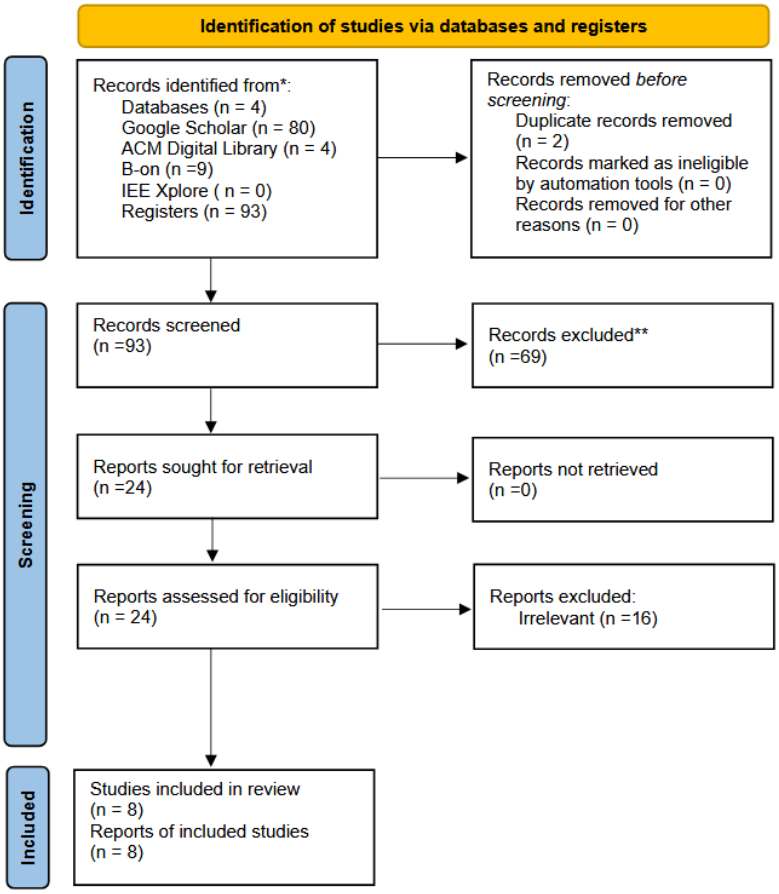

Figure 2 - Prisma systematic methodology
Adapted from [46]

The following table contains the analysed studies to answer the research questions.

Table 3 - Search results

Research Questions	Studies
RQ1	[4], [5], [11], [12], [13], [32], [47]
RQ2	[11]

3.6 Discussion

Although Protocol Buffers are a well-established serialisation technology, their application in RESTful architecture remains uncommon. Most studies examining Protocol Buffers focus on overall performance and efficiency benefits rather than their specific use in REST-based systems.

Similarly, with the growing necessity for high-performance and resource-efficient systems, Protocol Buffers could gain broader adoption due to their high performance. As a result, the adoption of binary serialisation could be more widely embraced.

3.6.1 RQ1: How does using Protocol Buffers impact the performance of HTTP-based REST architectures?

Protocol Buffers are highly regarded for their high performance and efficiency, making them well-suited for use in high-performance systems. Unlike JSON, a text-based format, Protocol Buffers employs a binary serialisation approach that produces smaller payloads, faster parsing speeds, and better data compression. These features enable it to excel in applications with stringent latency, memory, or bandwidth requirements [11], [12], [13].

A comprehensive evaluation by Juan Cruz Viotti and Mital Kinderkhedia [5] demonstrated that schema-driven serialisation formats, such as Protocol Buffers, consistently outperform schema-less formats like JSON regarding space efficiency. Even when JSON was compressed, Protocol Buffers also compressed and maintained their superiority in reducing data size, highlighting their adaptability to constrained environments. This phenomenon was pointed out at [5], which have shown the different studies comparing different serialisation formats and what these studies concluded.

Audie Sumaray and S. Kami Makki [4] corroborated these findings by showing that binary formats, including Protocol Buffers and Apache Thrift, outperformed JSON in serialisation speed, deserialization speed, and payload size. This is particularly critical for mobile platforms where resources are limited, as smaller payloads and faster processing reduce the overall overhead.

Table 4 - Average serialisation time in ms
Reprinted from [4]

	XML	JSON	Protocol Buffers	Thrift
Book	22.842	4.177	2.339	2.315
Video	17.884	4.097	1.800	1.747

Table 5 - Average deserialization time in ms
Reprinted from [4].

	XML	JSON	Protocol Buffers	Thrift
Book	7.908	1.199	0.298	0.732
Video	6.7.4.2	0.755	0.197	0.310

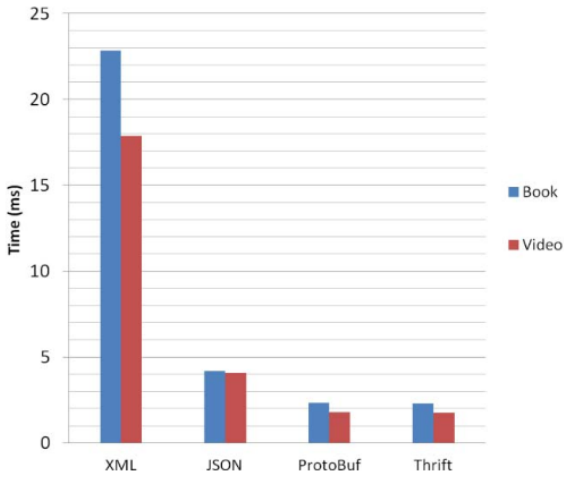

Figure 3 - Average serialisation time
Reprinted from [4].

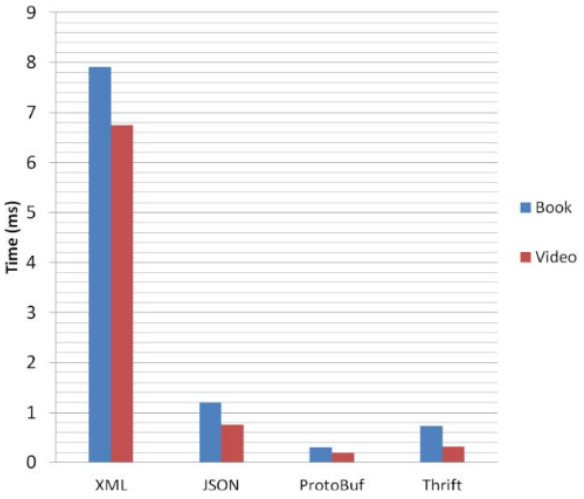

Figure 4 - Average deserialization time
Reprinted from [4].

From Table 4 and Table 5, a significant discrepancy in serialisation and deserialization speeds across the different formats can be observed. XML consistently demonstrates the highest average times for both serialisation and deserialization, indicating its relative inefficiency. That said, we can see binary formats such as Protocol Buffers and Thrift significantly outperform XML and JSON.

Protocol Buffers has the best performance for deserialization operations, slightly surpassing Thrift. This result aligns with its design goals of efficiency and compactness. JSON, while faster than XML, is notably slower than the binary formats, which is expected given its text-based structure.

Figure 3 and Figure 4 further corroborate these findings. In Figure 3, the serialisation time highlights the astonishing advantage of binary formats compared to text-based formats. Protocol Buffers and Thrift have near-identical performance, while XML struggles compared to all the other serialisation formats. Moreover, JSON is a middle ground between the other serialisation formats, providing moderate performance improvements over XML but failing to match the efficiency of Protocol Buffers and Thrift.

The study made by Eduard Maltsev and Oleksandr Muliarevych [47] quantified Protocol Buffers' efficiency, demonstrating an average payload size reduction of 33.06% compared to JSON. Such reductions have direct implications for network efficiency, enabling faster transmission and reducing storage requirements in systems with high data interchange volumes.

The most notable contribution to this field is the study by Vincenzo Buono and Petar Petrovic titled "Enhance Inter-service Communication in Supersonic K-Native REST-based Java Microservice Architectures"[32]. This research evaluates Protocol Buffers within the context of RESTful microservices, specifically targeting Quarkus-based, cloud-native architectures. Not only that, but it also highlights Protocol Buffers' advantages in serialisation efficiency and reliability, especially when computer resources are scarce.

Moreover, it demonstrated a significant performance improvement in serialisation processes, reducing response times by up to 25.1% in the best-case scenario compared to text-based formats like JSON. This reduction translates into faster request processing and an improved overall latency profile. Finally, it is revealed that a substantial decrease in payload size of 72.28% is achieved in the best-case scenario, this revelation further emphasises Protocol Buffers' capacity to optimise data interchange.

An observation from this research is Protocol Buffers' resilience in handling large or complex payloads, which JSON struggled with under memory-constrained conditions. The authors noted that JSON serialisation often failed to complete within allocated memory limits for highly nested or large data structures, whereas Protocol Buffers successfully serialised data, leveraging its efficient binary encoding.

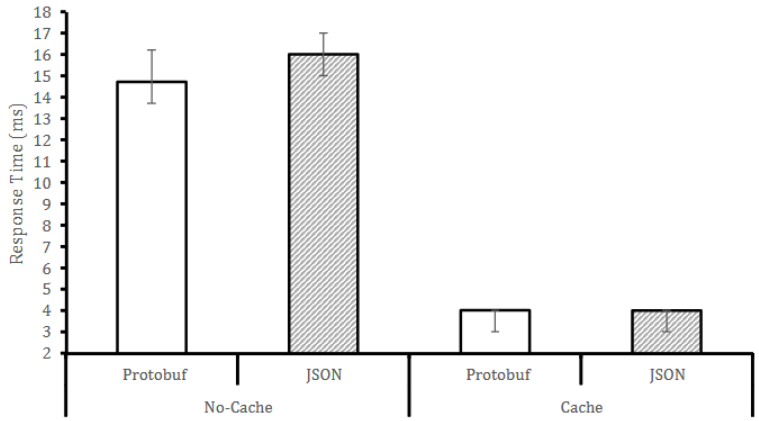

Figure 5 - Response time benchmark of a GET request uncompressed
Reprinted from [32]

From Figure 5, Protocol Buffers offer slightly better performance compared to JSON in the absence of caching. However, when caching is introduced, the difference in response time becomes negligible, demonstrating that caching mechanisms effectively mitigate any performance disparities between the two serialisation formats. This observation suggests that while Protocol Buffers are inherently more efficient, caching can serve as an equaliser under certain conditions, reducing the impact of serialisation inefficiencies on overall response time.

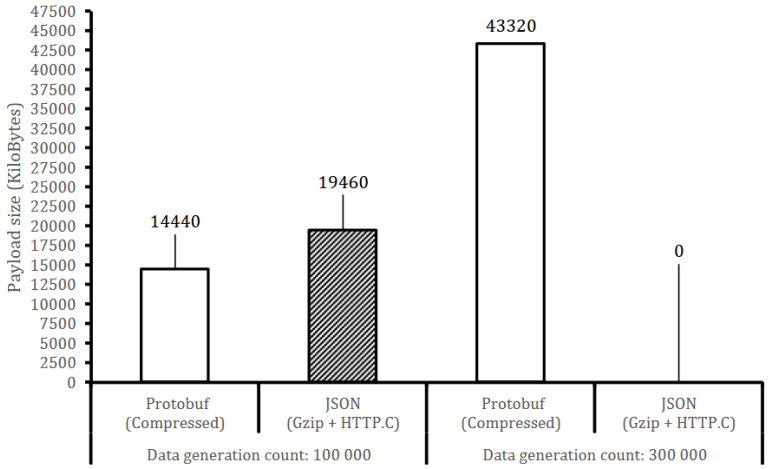

Figure 6 - Benchmark of payload size based on flat data
Reprinted from [32]

Figure 6 highlights the limitation of JSON, showing that when subjected to limited computer resources, it results in the termination of its process. This failure shows the limitations of JSON, especially in memory-constrained environments. Conversely, Protocol Buffers maintain their functionality under the same conditions, further solidifying their suitability for scenarios demanding high reliability and efficiency.

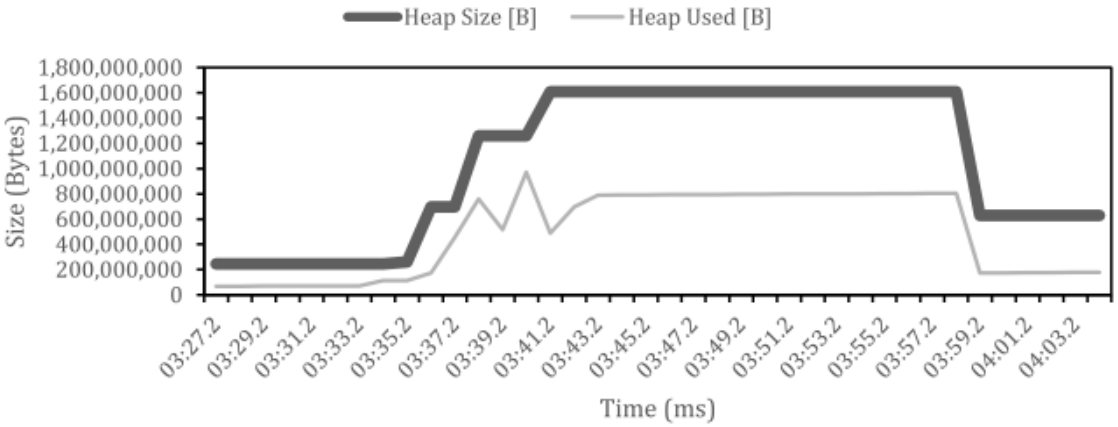

Figure 7 - Memory analysis of JSON serialisation
Reprinted from [32]

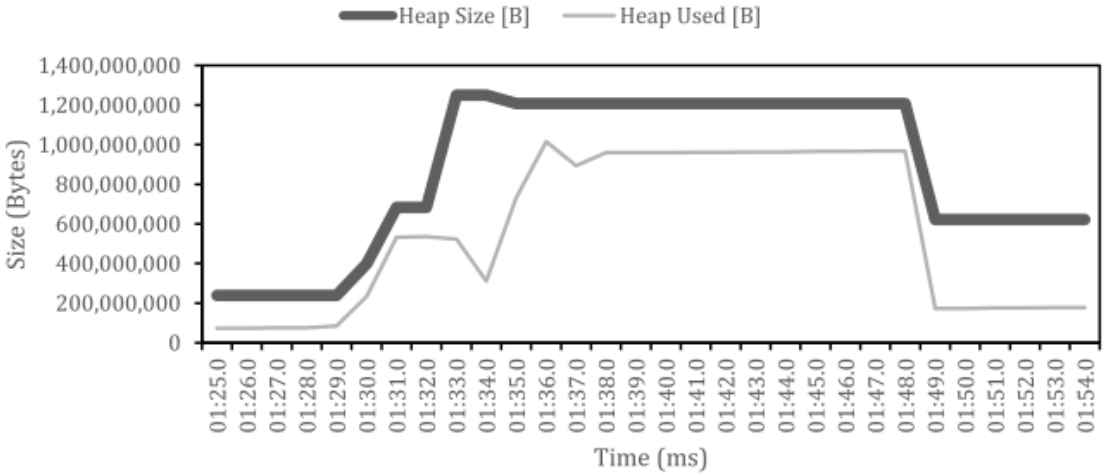

Figure 8 - Memory analysis of Protocol Buffers serialisation
Reprinted from [32]

Figure 7 and Figure 8 provide deeper insights into the memory profiles of the two formats during serialisation. Figure 8 reveals that Protocol Buffers consistently complete the serialisation process with significantly lower memory usage, showcasing their streamlined binary encoding mechanism when dealing with increased data count.

In contrast, Figure 7 demonstrates that JSON serialisation not only requires substantially more memory but also fails to complete the process in extreme cases. This inability to handle larger or more complex payloads in limited systems highlights JSON's lack of scalability compared to Protocol Buffers.

3.6.2 RQ2: To what extent can Protocol Buffers improve energy efficiency in HTTP-based REST architecture communications compared to JSON?

The study by Bruno Gil and Paulo Trezentos, titled "Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones,"[11] provides an analysis of the energy and performance implications of using three different data interchange formats, such as Protocol Buffers, JSON, and XML, on mobile devices.

Their findings revealed that Protocol Buffers are superior in terms of synchronisation time and energy efficiency, especially in large data volumes. On the other hand, the application of compression to text-based formats, such as JSON and XML, significantly narrows the performance gap.

Furthermore, the application of compression resulted in a reduction of approximately 66% in the size of text-based payloads, thus enhancing their performance on network interfaces of a lower data transmission speed, such as 3G, where the overhead of data transmission becomes more apparent.

Hence, when applications might require speed and raw efficiency, Protocol Buffers appear to be the superior choice. However, this efficiency comes with trade-offs, it was found that Protocol Buffers, while faster in synchronisation and processing, required more CPU energy for their operations compared to compressed JSON. This raises a critical consideration for mobile developers. Whether to prioritise faster data processing or minimise energy consumption, particularly in scenarios where battery life is paramount.

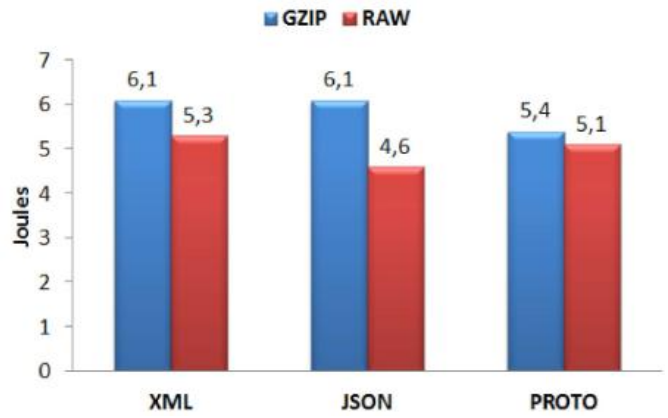

Figure 9 - Energy expended on CPU with data synchronisation (volume1)
Reprinted from [11]

Figure 10 - Energy expended on CPU with data synchronisation (volume2)
Reprinted from [11]

Both Figure 9 and Figure 10 show that Protocol buffers do indeed expend more energy on the CPU.

3.7 Conclusion

Protocol Buffers offer significant advantages over JSON in terms of performance and efficiency, particularly for applications requiring small payloads, low latency, and high throughput showing that binary serialisation reduces data size and improves processing speed, these benefits can be easily observed in environments where large data structures need to be transmitted reliably and efficiently.

However, Protocol Buffers also have problems that limit their applicability. In this case, their schema-driven approach adds complexity to development and maintenance. This problem is even more intense when data structures are constantly changing. Additionally, its binary format isn't human-readable, making it harder to debug and troubleshoot in comparison to JSON.

Furthermore, when resources are scarce, like in mobile applications, Protocol Buffers is shown to have increased energy consumption, which could outweigh its performance advantage over JSON. Because of that, Protocol Buffers can be better for applications that prioritise performance and scalability, nonetheless, JSON is still the preferred choice when simplicity and flexibility are required.

In conclusion, while Protocol Buffers offers clear performance benefits, its adoption still needs careful consideration, weighing its efficiency gains against the added complexity and resource trade-offs. As a result, Protocol Buffers can represent a promising alternative to JSON in HTTP-REST Based architectures.

4 Planning

The present chapter delineates the plan employed for the research, comprising five primary sections. The initial section pertains to the project charter, which is a dynamic document that encompasses information regarding the stakeholders, the scope of the project, the anticipated benefits, the deliverables, the project timeline, financial estimates, assumptions, constraints, and potential risks.

In addition, there is a work breakdown structure section and its dictionary, which describe the work required to ensure the successful conclusion of the project. In conclusion, the final two sections of the document, the timeline and the skills, illustrate the tasks from the work breakdown structure displayed in a timeline, and the skills that were required to complete the project.

4.1 Project Charter

The ensuing sections delineate the components that are generally present in a project charter, which is a living document that represents the project's preliminary planning. The project charter has the stakeholders, the scope of the project, the benefits of the project, the deliverables, the costs and the risks.

4.1.1 Stakeholders

This project provides valuable insights for diverse stakeholders, who may benefit in several ways from the knowledge presented. Table 6 outlines the defined stakeholders.

Table 6 - Power and interest matrix

Name	Power	Interest
Developers	Low	Medium
Software development companies	Medium	High
Students	Low	Medium
Researchers	High	High
Advisor	High	High

Each role has its gains from this work. For developers, this study provides a valuable source of knowledge regarding serialisation technologies, thus facilitating the development of more sophisticated technical decision-making skills. For software development companies, the insights gathered here might show an opportunity to reduce costs and improve the performance of their software.

For the students and researchers, this dissertation is intended to serve as a learning resource offering a comprehensive insight into the concepts of serialisation and their impact on software.

One side note, more specifically for researchers, this may serve as a source of inspiration for the development of new technologies and the initiation of further academic studies.

Finally, the advisor's role is to ensure that the project aligns with academic standards, offering guidance and assistance in overcoming challenges.

4.1.2 Scope, Objectives

Nowadays, in software engineering, the efficiency of data serialisation has a direct influence on the performance of REST architectures. JSON, as the dominant serialisation format, is highly regarded for its simplicity and human readability. However, it has some performance issues, such as larger payloads and slower parsing, making it less suitable for resource-constrained or high-throughput scenarios.

In such cases, a possible solution would be to utilise an alternative serialisation format, like Protocol Buffers, which offers reduced payload sizes and faster parsing speeds, which is a consequence of its binary serialisation format. However, issues regarding its complexity and the challenge of readability frequently result in limited adoption.

Moreover, the primary objective of this thesis is to perform a comprehensive evaluation of Protocol Buffers in a REST architecture and compare key metrics with the same one with JSON. There are performance metrics and energy consumption metrics that are analysed.

In terms of performance metrics, such as the average response time, which is the average time it takes for the system to respond, the median response time, the throughput, which is the number of completed requests per unit of time, and the maximum and minimum response times are analysed.

Furthermore, concerning energy consumption, the key metrics are the average energy consumption, which is the mean energy consumed during the testing period, the median, the maximum value, the minimum value, and, ultimately, the joules per request, representing the energy expended for a single request.

The insights gathered are intended to assist developers and software architects with reasonable evidence on whether to adopt a determined technology in a REST architecture, like JSON or Protocol Buffers.

4.1.3 Benefits

The present study has certain advantages, the primary one being the possibility of cost reduction, which, if demonstrated, that Protocol Buffers are more efficient than JSON could lead to a reduction in bandwidth and operational costs. Consequently, this development may spur the research and innovation of new serialisation technologies, aimed at enhancing performance and energy efficiency.

4.1.4 Deliverables

The project includes a project plan, which delineates the dissertation's timeline and task breakdown, which is a list of tasks and their respective completion times. It also contains the dissertation report, which is the document that comprises the study, techniques, findings, and interpretations.

Another important element is the software code, encompassing all code related to the experimental setup, including benchmarks and test scripts. These can be found in the project repository [48], which contains an MIT license.

The final deliverable is the presentation document, which is a summary of the project's key findings.

4.1.5 Time

These are the following milestones and mandatory dates for the project.

Milestones:

- Goal Question Metric (GQM) -15/01/2025
- Hypothesis Test - 24/01/2025
- Control Project - 13/02/2025
- Data From Control Project - 14/03/2025
- Changed Project - 14/03/2025
- Data from changed project - 09/04/2025
- Finished analysis - 09/04/2025
- Report 1.0 delivery - 17/04/2025
- Final Report Delivery - 29/06/2025

Mandatory Dates:

- Prepd review delivery - 06/12/2024
- Report and Presentation Delivery - 04/01/2024
- Final Delivery - 29/06/2024

4.1.6 Costs

No direct costs were identified for this project, as it relies on open-source tools (e.g., JMeter and Kepler, which were already presented before in the background section), institution-provided infrastructure, and digital libraries accessible through academic licensing. However, indirect costs such as time investment and a learning curve for new tools and methodologies were expected.

4.1.7 Assumptions, Restrictions and Risks

During this project, certain assumptions were made, such as the assumption that the benchmarking tools and required libraries were consistently accessible and that the experiment setup was always consistent across tests. Another one is consistency across tests involves maintaining the same environment and test specifications.

In addition, certain restrictions were implemented, including the use of real-world datasets. This was done to ensure compliance with ethical requirements and to restrict the use of real data to synthetic data exclusively. A further limitation is the scarcity of resources and time, which restricts the scale of the experiments and the extent of the analysis conducted.

The final restriction implemented involved the necessity of conducting tests in a controlled environment, adhering to the selected research methodology [23].

Another important component is the risk management which was done to ensure that potential threats were identified, assessed, and mitigated, this ensured that there wouldn't be any disruptions or compromises in the project timeline, consequently, creating a Risk Register was a top priority to document and monitor risks throughout the project lifecycle, it identifies key risks, their causes, and the planned responses.

The Risk Register highlights 9 primary risks (Figure 13 and Figure 14 in Appendix A), covering various aspects of the project, including tool compatibility, data quality, and resource availability. Each risk has been evaluated for its probability and impact, or in short, PI, resulting in a prioritisation based on the probability and impact score [49].

Finally, mitigation strategies have been outlined for each risk to ensure project continuity and success. The Risk Register is a living document that is reviewed and updated periodically as new risks emerge or existing risks evolve.

4.2 Work Breakdown Structure

The work Breakdown structure is in Appendix A, Figure 15 , and it is divided into three main phases. The experimental Design and Setup involve defining the research methodology, establishing the experimental environment, and collecting baseline data. It sets the foundation for the comparison between JSON and Protocol Buffers.

The Documentation and Reporting phase, where data from the experiments are analysed to identify trends, conclude, and address the research questions, the findings are then documented in a structured format to form the core of the dissertation.

Finally, we have the final phase focusing on refining the dissertation based on feedback, preparing the presentation, and ensuring all deliverables are submitted on time. It concludes with the final presentation and evaluation of the research.

4.3 Work Breakdown Structure Dictionary

This section provides a comprehensive examination of the detailed work breakdown structure (WBS) dictionary. Table 26 is a dictionary in which each row corresponds to a phase, deliverable or work package. Each entry is accompanied by a description and progress criteria. Empty descriptions are self-explanatory and require no further elaboration.

4.4 Timeline

The complete timeline is presented in Appendix A Figure 16. It should be noted that the image was divided into two parts, and these are divided into Figure 17 and Figure 18, which can also be found in Appendix A.

4.5 Skills

This chapter is concerned with an analysis of the competencies necessary for the successful completion of the project. It also includes an evaluation of strengths and weaknesses and the formulation of strategies to address the latter. Furthermore, the chapter also includes an assessment of technical skills.

4.5.1 Required Skills

To perform this research successfully, the project demands a diverse set of skills like analytical thinking and problem-solving, lifelong learning, time management, and critical thinking, while additional skills may also play a role, the ones stated are important to ensure the project's success.

4.5.2 Strengths and Weaknesses

The development of the research has been influenced by a self-reflection that identified qualities that contribute to the development of this project. The first of these qualities is the desire to constantly learn, and the second is the ability to adapt to problems.

A notable challenge that must be addressed concerns my writing abilities, which, in general, constitute a significant hindrance to the main part of the thesis. This aspect demands that I be able to articulate my thoughts and conclusions in a coherent and comprehensible manner through written text, while also being able to effectively communicate these findings to others.

4.5.3 Technical Skills

As the work to be done requires some technologies that I have never used. Tools such as JMeter and Kepler were previously talked about in the background section, but Protocol Buffers is also something that I needed to have a better understanding of. The following table displays the technical skills needed to complete the dissertation successfully.

Table 7 - Technical Skills needed to complete the dissertation

Skill	Required Proficiency	Current Proficiency	Comments
Springboot	8	7	Springboot might have some features that are destined for Protocol Buffers and might require some review.
Java	9	8	There might be some useful functionalities in it that might prove useful for Protocol Buffers, and these may require some attention and search.
Gradle	4	2	Some knowledge comes from the Jenkins pipeline and, as such, is basic and might require some further reading of the documentation.
Protocol Buffers	10	4	Somewhat of a new concept for me that required further reading of documentation and articles to bring the most out of the tool.
JSON	10	10	
Docker	5	5	
Kubernetes	7	5	Decent knowledge of Kubernetes, but never used it to gather energy consumption, so it requires further learning.
Grafana k6	9	1	The needed knowledge for JMeter has already been worked on before, and most of the features are already known.
Kepler	9	1	Basic understanding of the tool. Also, another one on which I have never worked.

Table 7 shows the needed skills to be able to complete the dissertation, it is given to each technical skill a required/current proficiency that goes from 0 to 10 to assess my capabilities and work to be done for that specific skill.

4.5.4 Plans for Improvement

As demonstrated in the preceding sections, improvements must be made if this work is to be completed. For this reason, an improvement plan has been devised for each area.

The weakness to be addressed is my writing, for which I utilise tools such as Grammarly to refine my grammar and expand my vocabulary. In addition, the intention was to review scientific articles in areas of personal interest, to enhance writing skills and cultivate critical thinking.

Finally, in order to enhance my technical capabilities, a substantial amount of documentation study and practical application of the relevant technologies was performed. This enabled me to develop proficiency with tools with which I had less familiarity, whilst simultaneously addressing any knowledge gaps with already known tools.

5 Experiment

This chapter focuses on a brief analysis of the selected project migrated to Protocol Buffers, it outlines why the project was chosen, what changes were made to the base project, what was migrated to Protocol Buffers, and why these specific parts of the system were migrated.

The present chapter is concerned with the performed experiment, from the project selection to the conclusions derived from the data collected. The project is comprised of five primary sections. The initial section is the project selection, which delineates a set of criteria for the selection of a project for the experiment.

In the subsequent section, the experiment preparation, a comprehensive description is provided of the complete experimental setup, together with the contextualisation of the project's architecture and its components. It also outlines the strategy for modifying the project.

Moreover, the migration section provides a comprehensive explanation of the plugins and dependencies utilised, the migrations, and the configuration for the Kubernetes cluster.

Finally, the analysis and conclusion sections of this study provide insight into the gathered data from the experiment through statistical analysis and a hypothesis test.

5.1 Project selection and analysis

The chosen project is a convenience store HTTP REST API developed in Java with the framework Spring Boot, which can be found on GitHub [50]. The following criteria were followed to choose a decent project that would make sense for the research being developed.

- The project needs to have commits within the last year, so any project with commits during or after February 2025 can be accepted.
- The project must have an MIT license, as it allows unrestricted utilisation of the project and authorises individuals to exercise full autonomy in their engagement with it.

- The project should, preferably, use an in-memory database, this implies that external factors, like network or overhead in I/O, don't introduce variability, which is expected when performing experiments in controlled environments.
- The project can't be too simple nor too complex, the main objective is to focus on performance and energy efficiency, and, as such, there should be at least some endpoints that allow for a decent comparison.

The Spring Boot project in question is a REST API designed to manage a convenience store, with a range of products, users and transactions. The project was selected because it is neither complex nor simplistic, and so it possesses sufficient entities to enable a satisfactory benchmark.

In addition, the project is well-structured, enabling a faster comprehension of its architecture and implementation. It has an in-memory database, which is a significant advantage, and its most recent commit occurred less than a year ago. In essence, the project fulfils all the outlined criteria.

To maintain impartiality regarding the project, it was deemed necessary to implement changes that were as minimal as possible. One of the earliest modifications was the database. The original in-memory database, HSQLDB, was not easily accessible for debugging. Consequently, the H2 in-memory database was implemented as a more accessible alternative.

The second and final modification entailed the incorporation of the Prometheus dependencies and the actuator dependencies. This integration was implemented to enable the extraction of metrics from the HTTP REST API by Prometheus.

In addition, to maintain the integrity and impartiality of the systems, both APIs were subjected to integration tests. These tests ensured that the return values were consistent and that the implementations remained functional.

This testing was achieved through a custom bash script that can be found at the project repository [48] in the test directory starting from the root. It tests the used endpoints for the performance tests with predefined data, where there are expected files, which is data expected when performing certain requests, and it has the normal data file, for POST or PUT requests.

Moreover, this is a rudimentary testing framework developed with the principal objective of integration testing. The decision to create this framework was driven by the absence of tools capable of sending pb or bin files through REST.

Finally, both APIs were tested at the start and the end of the setup. Not only that, but after the implementation in Kubernetes, both APIs were tested to ensure consistency and their respective functionality.

5.2 Experiment preparation

This section includes the selection of the components to be migrated, why they were chosen and what the thought process was for the migration of the API.

It is important to note, once again, that, to maintain the relevance of the case study being developed, it is of the utmost importance to not change any logic of the software, not only that, but if modifications are deemed necessary to the new implementation to work properly, it needs to be specified what was changed and why. This is crucial because we want the project to remain as close as possible to the base implementation so that we do not have too many confounding variables that impact the analysis being conducted.

5.2.1 Experimental Setup

To promote the reproducibility of the obtained values, a few details about the testing conditions need to be mentioned, in this case, the software requirements to run the tests, the hardware on which the tests were run, the networking conditions in which the tests were run, and finally, the scripts utilised during or to setup the tests.

To execute the tests, the following software prerequisites were necessary, Minikube [51], which is a local Kubernetes tool to run a single-node Kubernetes cluster, Helm [52] which is a package management tool for Kubernetes, Grafana k6 [40], which serves to execute performance tests and finally Nodejs [53], which is a JavaScript runtime to run the k6 test scripts.

There are two important notes to make. Firstly, in the event of a decision to construct the entire system from the ground up, it is required to install Docker, Docker Compose and Protoc, the latter of which is the compiler for Protocol Buffers. Furthermore, to operate the Java API, Maven and Java are also required. Secondly, in Appendix B, Table 27 provides a list of the software requirements and their respective versions.

Furthermore, the hardware specification where the tests were run is shown in Table 8, which contains the operating system, the processor, the memory and the disk of the machine.

Table 8 - Testing hardware specification

Hardware	**Description**
Operating system	Ubuntu 24.04.2 LTS
Processor	12th Gen Intel Core i5-1240p x 16
Memory	16 GB
Disk	512.1 GB

Moreover, with regard to the networking conditions, localhost was utilised, given that the Kubernetes cluster was local, with each service being port forwarded in order to enable the scripts to reach their targeted service.

Finally, a set of bash scripts was utilised to automate and minimise errors during data extraction or even service execution. The first script is “automate_execution.sh”, which allows the startup of Minikube, installs the necessary packages with helm and ports forward the necessary services for tests.

The second script utilised is "run_k6_tests.sh", which traverses each k6 script file and executes the test, subsequently extracting the energy consumption of the test that has been executed. To execute the standard tests, it is necessary to pass the bearer token as an argument. Furthermore, to test the API gateway, as discussed in greater detail in the current chapter, the "--gateway" flag must be specified.

Finally, all extracted data, the k6 scripts, the bash scripts, and the data analysis scripts can be found in the project repository [48].

5.2.2 Architecture

The initial stage of the project was just a simple Springboot API with a connection to an in-memory database. Figure 11 shows the component diagram from the base structure of the convenience store API from [50].

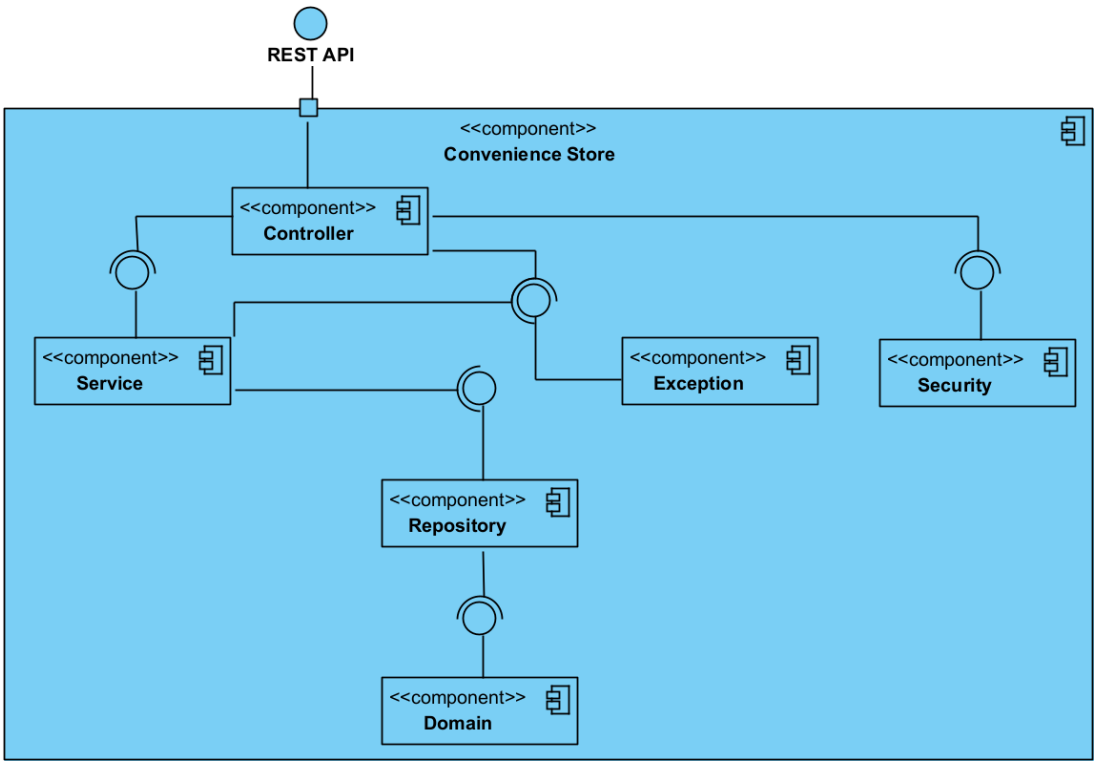

Figure 11 - Component diagram

The accompanying diagram illustrates a standard Spring Boot API, incorporating the domain component, which consists of the domain classes connected to the business. Subsequently, the repository component facilitates database requests through its Java Persistence API (JPA) abstraction. This is followed by the service component, which encompasses the business logic surrounding the domain.

Another component is the security component, which provides authentication functionality via JSON Web Token (JWT) and the Spring Boot security dependency. Furthermore, the exceptions component oversees the management of exceptions arising within the services and controller layers. The controller component is responsible for processing incoming HTTP requests and generating the appropriate responses.

Furthermore, one of the questions asked at the beginning of the work was what would be lost when using Protocol Buffers. One of them was the API documentation through OpenAPI [54], which is a standard to describe HTTP APIs, this specification doesn't work out of the box with Protocol Buffers, as such, making it one possible impediment to using Protocol Buffers in an HTTP REST architecture.

Nevertheless, it remains feasible to create custom definitions within the API for the specific endpoints that utilise classes from proto files. The objective is to enable the creation of documentation that aligns with the Protocol Buffers specification in JSON. The potential exists to enhance this system even further. In a practical setting, a complete overhaul of the system is improbable, necessitating a gradual transformation.

To this end, the implementation of an API gateway is proposed, a strategy that utilises both documentations while ensuring minimal performance degradation. This approach maintains the accessibility of the legacy system in JSON with the documentation, while concurrently introducing the new system with Protocol Buffer and its customised documentation.

As illustrated in Figure 19 in Appendix B, the proposed architecture's deployment diagram is displayed. The subsequent diagram illustrates the API gateway, which exposes a Swagger UI with documentation for both APIs. It is important to note that each API, including the gateway, possesses its unique namespace, allowing for an easy extraction of metrics through Prometheus, which can isolate namespaces and extract data based on just these namespaces.

Finally, regarding Figure 19, Prometheus and Grafana were a decent option in the beginning to debug and extract certain metrics. However, they had some problems, specifically with time frames to extract energy consumption metrics, as such only the Kepler endpoint for metrics was used. Even though the tools mentioned before became somewhat obsolete, they remain a great way to debug the systems in the Kubernetes environment.

5.2.3 Components

The process of selecting a component essentially consisted of interpreting the topic of the dissertation, like selecting components for which the serialisation method was more intensive, so that their weight would be more significant in terms of performance, and conclusions could then be drawn.

For example, cases in which objects are returned within objects, or objects with lists of objects, are especially good for this analysis. However, components that exhibit less intensive serialisation were also selected to ascertain whether, in all situations, one serialisation method consistently outperformed the other.

The primary change implemented was to modify the Users domain. This alteration was necessary because it was identified as one of the entities with the most fields requiring a lot of serialisations, making the serialisation mechanism important for this process.

However, it is interesting to analyse the impact of serialisation on requests that need more processing power, particularly in the case where hashing is required, which is a process that demands significant computational resources. This analysis could potentially contradict the hypothesis that a faster serialisation format would always offer a performance advantage, suggesting that the complexity of the process may outweigh the benefits of the format itself.

The final component to be migrated was the Product Entities domain. This component was selected on the basis that it would contain one of the largest amounts of data to be transmitted, thus allowing a comprehensive evaluation to be conducted to ascertain the relative merits of one serialisation format in comparison to another.

5.2.4 Strategy

The idea was to initiate the process at the Data Transfer Objects (DTOs), which function as the primary interface for the serialisation process, and then proceed sequentially, starting from the Controllers and culminating at the Service level. The implementation process is elucidated in the subsequent section.

6 Migration

The present section is concerned with the implementation details of the solution using Protocol Buffers, and it demonstrates the various dependencies that were utilised, the testing tools and their configuration, the reason they were employed, and other processes during this implementation.

6.1.1 Dependencies and Plugins

A few dependencies and plugins were needed for Protocol Buffers to work properly, one of the first dependencies added to the pom.xml file is the Protocol Buffers-java dependency, which allows us to use and process Protocol Buffers inside the API, furthermore, in Code Snippet 5, it was created a bean that allows for the requests and responses to be sent as a Protocol Buffers with that same dependency.

```
package com.conveniencestore.conveniencestore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.http.converter.Protocol Buffers.Protocol
BuffersHttpMessageConverter;

@SpringBootApplication
public class ConvenienceStoreApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConvenienceStoreApplication.class, args);
    }


    @Bean
    Protocol BuffersHttpMessageConverter Protocol
BuffersHttpMessageConverter(){
        return new Protocol BuffersHttpMessageConverter();
    }
}
```

Code Snippet 5 - Changes to allow Protocol Buffers in the project

After this, a plugin was added that would compile the Protocol Buffers files (.proto) into the Java classes without having to manually compile them to make it even better, another plugin was added, which allows Maven to get the relative paths of the project, which allows anyone to compile the Protocol Buffers files without trouble.

```
<build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
                <version>1.7.1</version>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>Protocol Buffers-maven-plugin</artifactId>
                <version>0.6.1</version>
                <extensions>true</extensions>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>test-compile</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>

<protoSourceRoot>${project.basedir}/src/main/proto</protoSourceRoot>
                    <protocArtifact>com.google.Protocol
Buffers:protoc:4.29.3:exe:${os.detected.classifier}</protocArtifact>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Code Snippet 6 - Dependencies to compile Protocol Buffers automatically

As illustrated in Code Snippet 6, the employed plugins were the os-maven-plugin and the Protocol Buffers-maven-plugin. The former enables Maven to store the relative path of the project in a variable, designated here as ${project.basedir}. The second plugin utilises this feature to identify the path to the Protocol Buffers files. Following the specification of the

desired execution lifecycle, in this case, the compile lifecycle, the plugin compiles those proto files into Protocol Buffers Java classes.

Finally, a plugin for IntelliJ IDEA was used to automate the conversion of plain old Java objects (POJOs) into proto files. In this instance, the plugin was applied to DTO records, which do not contain business logic and are primarily used for data transfer. The decision to undertake this step was motivated by ethical considerations, namely, to refrain from altering the primary code in a manner that was not intended.

```
package com.conveniencestore.conveniencestore.domain.users;

import java.time.LocalDateTime;

public record UserResponseJsonDTO(
        Integer id,
        String username,
        String email,
        UserRoles role,
        LocalDateTime createdAt,
        LocalDateTime updatedAt

) {
}
```

Code Snippet 7 - Generated user response DTO

```
syntax = "proto3";
import "user_roles.proto";
import "local_date_time_pb.proto";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message UserResponseDTO {
  int32 id = 1;
  string username = 2;
  string email = 3;
  UserRoles role = 4;
  LocalDateTimePb created_at = 5;
  LocalDateTimePb updated_at = 6;
}


message UserResponseCatalog {
  repeated UserResponseDTO users = 1;
}
```

Code Snippet 8 - Generated user response Protocol Buffers file

As illustrated in Code Snippet 7, the record utilised to generate the proto file is depicted in Code Snippet 8. However, several elements were absent from the record, including the UserResponseCatalog, the LocalDateTimePb, and the UserRoles. Additionally, the automatic generation of the package definition isn't possible, also, the explicit usage of proto3 syntax was done manually.

The reason for the absence of generation of the protos that are being imported into the specified proto file is the absence of a native representation of LocalDateTime in Protocol Buffers, as is the case in Java. Therefore, a new proto file is required for this purpose. A similar situation arises with ENUMS, in this case, the User. The User enum is generated externally and subsequently imported.

Finally, the UserResponseCatalog is employed to represent a list of User responses. This is because the generated Protocol Buffers classes do not support being sent as a List and must be pre-declared in the same manner as it is in that proto file.

6.1.2 User Domain Migration

As previously stated, the initial step involves the conversion of every DTO into a proto file. The procedure for this conversion is outlined in Section 6.1.1, where the utilisation of an IntelliJ plugin is demonstrated. These files are in the src/main/proto directory. Appendix D illustrates Code Snippet 19 to Code Snippet 22, which depict the generated proto files associated with the employed User DTOs.

To compile these proto files to Java, it is necessary to execute the compilation lifecycle action in Maven. The files in question were generated in the directory designated as target/generated-sources/Protocol Buffers/java/com/conveniencestore/conveniencestore/Protocol Buffers.

```
 @PostMapping(produces = "application/x-Protocol Buffers", consumes =
"application/x-Protocol Buffers")
    public ResponseEntity<?> registerNewUser(@RequestBody @Valid
UserDto.UserDTO data) {
        if (data.getPassword().isEmpty()) {
            ErrorDTO error = new ErrorDTO("Please provide the password.",
400);
            return ResponseEntity.status(400).body(error);
        }
        UserResponseDto.UserResponseDTO user = this.service.insert(data);
        return ResponseEntity.ok(user);
    }
```

Code Snippet 9 - Example of one endpoint that uses Protocol Buffers

As demonstrated in Code Snippet 9, the necessary modifications to use Protocol Buffers can be identified. In this case, we need to specify what it produces and what it consumes. In this case, we specify the following “application/x-Protocol Buffers”.

Furthermore, to transmit a request body with a Protocol Buffer, it is necessary to specify the data that will be serialised. In this case, the UserDTO class, which was generated with the proto file, is used. A minor detail from Code Snippet 9 is that we refer to UserDTO.UserDTO, the first part is the outer class generated by the compiler, as no class was specified in the proto file, and thus it remained as the name of the file (User_dto.proto).

Finally, a UserResponseDTO is returned because of the insertion. If the insertion is successful, the same principle applies to the outer class, explaining the code's verbose nature. An

important aspect of this endpoint is that the ErrorDTO was not changed to Protocol Buffers, this is simply because errors were not expected to occur during the benchmark, nor was it intended to be analysed. As such, it was not changed to a proto class.

```
public UserResponseDto.UserResponseDTO insert(UserDto.UserDTO data) {
        if (userRepository.findUserByEmail(data.getEmail()).isPresent())
throw new UserAlreadyExistsException();
        String password = new
BCryptPasswordEncoder().encode(data.getPassword());
        data = UserDto.UserDTO.newBuilder()
                .setUsername(data.getUsername())
                .setEmail(data.getEmail())
                .setPassword(password)
                .setRole(data.getRole()).build();
        User user = new User(data);
        user = this.userRepository.save(user);

        return UserResponseDto.UserResponseDTO.newBuilder()
                .setId(user.getId())
                .setUsername(user.getUsername())
                .setEmail(user.getEmail())
                .setRole(convertRole(user.getRole()))
                .setCreatedAt(convertLocalDateTime(user.getCreatedAt()))
                .setUpdatedAt(convertLocalDateTime(user.getUpdatedAt())).bu
ild();
}
```

Code Snippet 10 - Example of service level insert method using Protocol Buffers

Lastly, in the user domain. It's also straightforward, the Protocol Buffers are simply sent through the signature of the method, and the method's tasks are then performed. As illustrated in Code Snippet 10, a user is inserted, as demonstrated, there is a considerable amount of verbosity. This is particularly evident in the serialisation and deserialisation processes, which are executed multiple times. It is noteworthy that the logic employed in this domain is analogous to that used in the control project.

The primary objective was to preserve the existing logic while enabling the conversion of JSON. A notable aspect is the usage of helper functions that enable the conversion of user fields into their designated data types, such as ConvertLocalDateTime and ConvertRole. The incorporation of these functions was essential to enhance the readability of the code. The display of these helper functions can be observed in Code Snippet 11.

```
private UserRolesOuterClass.UserRoles convertRole(UserRoles role){
        if (role == UserRoles.ADMIN) return
UserRolesOuterClass.UserRoles.ADMIN;
        else return UserRolesOuterClass.UserRoles.EMPLOYEE;
    }
    private LocalDateTimePb convertLocalDateTime(LocalDateTime time){
        return LocalDateTimePb.newBuilder()
                .setYear(time.getYear())
                .setMonth(time.getMonthValue())
                .setDay(time.getDayOfMonth())
                .build();
    }
```

Code Snippet 11 - Helper methods to convert special data types into Protocol Buffers

As previously indicated, this process was repeated throughout the user domain. First, the necessary DTO was converted into a proto file, and then it was compiled into a Java class. After that, changes were made at the controller level, and finally, at the service level.

6.1.3 Product Entity Domain Migration

The procedure to generate a proto file is replicated. Initially, proto files are created. Appendix D contains the generated proto files for this domain, as illustrated in Code Snippet 23 to Code Snippet 25. Subsequently, the Java classes are generated by the Protocol Buffers compiler.

```
@GetMapping(produces = "application/x-Protocol Buffers")
    public ResponseEntity<?> getAllProducts(
            @RequestParam(required = false, defaultValue = "id")
            String orderby,
            @RequestParam(required = false, defaultValue = "asc")
            String order
    ) {
        if (VALID_SEARCH_PARAMETERS.contains(orderby) &&
VALID_SEARCH_PARAMETERS.contains(order))
            return
ResponseEntity.ok().body(ProductEntityOuterClass.ProductEntityCatalog.newBu
ilder().addAllProducts(this.service.getAll(orderby, order)).build());
        ErrorDTO error = new ErrorDTO("Request param is not valid.", 400);
        return ResponseEntity.status(400).body(error);
}
```

Code Snippet 12 - Example of one endpoint for the Product Entity domain

The same principle is applied here, as in the user domain. As illustrated in Code Snippet 12, there are changes required to use Protocol Buffers, in this case, a GET request is made for a list of Product Entities. Protocol Buffers do not support Java lists as a native type. Consequently, a specialised proto is required for the list.

As illustrated in Code Snippet 23, ProductsEntityCatalog is essentially a list of ProductEntities. The creation of ProductsEntityCatalog is done by simply calling the builder method and adding all the Product Entities with the addAllProducts method.

```
public List<ProductEntityOuterClass.ProductEntity> getAll(String orderby,
String order) {
        Sort.Direction direction;
        switch (order) {
            case "asc" -> {
                direction = Sort.Direction.ASC;
            }
            case "desc" -> {
                direction = Sort.Direction.DESC;
            }
            default -> {
                direction = Sort.DEFAULT_DIRECTION;
            }
        }
        return productEntityRepository.findAll(Sort.by(direction,
orderby)).stream().map(this::convertFromProductEntity).toList();
    }
```

Code Snippet 13 - Example of service level method to retrieve all Product Entities with Protocol Buffers

As illustrated in Code Snippet 13, an example of a method migrated to Protocol Buffers in the Product entity domain at the service level is shown.

6.1.4 Kubernetes Cluster Setup

In order to facilitate the seamless execution of tests and the utilisation of Kepler, a Kubernetes cluster was configured for both the control project and the experimental project. To this end, a Docker file was created for each API, subsequently published on Docker Hub.

This publication was facilitated through the pipeline in Jenkins, which traversed each directory, constructed the Java Archive (JAR), which is an executable file that can be run in any environment, and then converted it into a Docker image. These images were then exported to their respective Docker repositories [55], [56]. For reference, Code Snippet 14 contains the Docker file for both projects.

```
FROM openjdk:17


COPY target/convenience-store-1.0.0.jar convenience-store-1.0.0.jar

EXPOSE 8080

ENTRYPOINT ["java","-jar","convenience-store-1.0.0.jar"]
```

Code Snippet 14 - Docker file for both projects

After this, each project is given a Kubernetes directory, called k8s, with its respective service, deployment and namespace creation for the cluster. It is recommended to use Minikube [51], as it significantly simplifies the usage of Kubernetes. Not only that, but before running the projects, Minikube should be started, and Kepler, Prometheus and Grafana need to be set up beforehand, for which Kepler has all the necessary documentation [52].

The configuration of the YAML Ain't Markup Language (YAML) manifests for the applications is relatively straightforward, however, the most important part is the one shown in Code Snippet 15, which tells Prometheus to monitor the exporter from our API. Additionally, the configuration encompasses the monitoring of memory and CPU usage in its deployment manifesto, as illustrated in Code Snippet 16.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: experimental-project
  namespace: app-namespace
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: experimental-project
  endpoints:
    - port: web
      path: /actuator/prometheus
      interval: 15s
```

Code Snippet 15 - Service monitor manifest file for API

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: experimental-project
  namespace: app-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: experimental-project
  template:
    metadata:
      labels:
        app: experimental-project
    spec:
      containers:
        - name: experimental-project
          image: 1230199/experimental_project:latest
          ports:
            - containerPort: 8080
              name: web
          resources:
            requests:
              memory: "512Mi"
              cpu: "250m"
            limits:
              memory: "1Gi"
              cpu: "700m"
```

Code Snippet 16 - Deployment manifest file for API

It was deemed necessary to automate the repetitive and tedious configuration required to run the tests, for which purpose a bash script was created. The script initiates the Docker process,

followed by the initialisation of a Minikube cluster. The script then installs the necessary dependencies, applies the manifests, and subsequently ports forward the essential services.

The script in question can be located in the designated "scripts" directory, named "automate_execution.sh", and must be executed from the project's root directory.

6.2 Analysis

This section provides insight into the experiments developed and the methodology used to evaluate the comparison between JSON and Protocol Buffers. Furthermore, the GQM approach [57] is then used to establish metrics for performance and energy consumption. One important note is the necessity to have a controlled environment to perform this experiment, which is important since there is a need to minimise confounding variables.

Firstly, a controlled environment was established for the execution of the tests. This is important because it minimises the confounding variables that can skew the data collection and compromise the given experiment. As previously mentioned, the tests were executed within a Kubernetes cluster, which is a controlled environment.

Additionally, the REST architecture was isolated in a distinct namespace from the various tolling mechanisms within the cluster. This fact was previously mentioned. Finally, it was already shown in Table 8 the hardware specifications of the utilised machine.

6.2.1 Goal Question Metric

The approach is composed of three primary components. The first component is the conceptual level, also referred to as the goal, which is the point we want to achieve with the methodology, the second component is the operational level, which is the set of questions derived from the goal we want to achieve, and finally, the third and last component is the quantitative level, which is the set of metrics to be collected to answer the questions [57] Figure 12 is a visual representation of how the GQM can look.

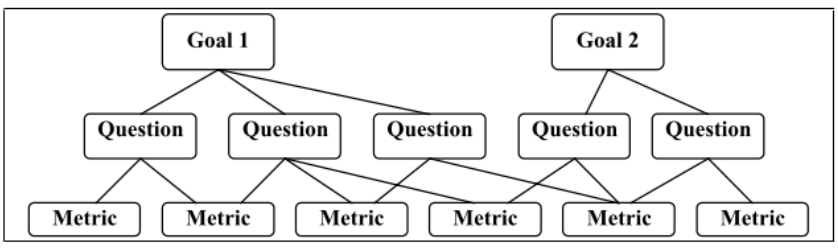

Figure 12 - Representation of a possible GQM approach
Reprinted from [57]

The central objective of this dissertation is to evaluate the performance of Protocol Buffers in comparison to JSON within the same HTTP REST architecture. Consequently, the goal and questions are derived from the topic and the research questions previously shown in Section 1.3, as such, the following goal was defined. “Analyse the impact of using Protocol Buffers versus JSON in HTTP based REST architectures to evaluate their performance and energy consumption”.

6.2.1.1 Performance

Table 9 - Question and metrics for performance

Question	**Metric**
What is the difference in energy consumption of Protocol Buffers against JSON in HTTP-based REST architectures?	Average response time (ms)
	Throughput (req/s)
	Median response time (ms)
	Max (ms)
	Min (ms)

To further evolve the GQM, it is essential to understand the metrics being evaluated, which were later compared between the two serialisation formats. According to Table 9, two metrics have been defined for measurement, in this case, throughput and response times. Throughput is the number of requests processed within a timeframe [58], which indicates the capacity of the API to handle a high volume of requests.

Finally, the response time is defined as the duration required to respond to a single request, which is important, as it helps understand the weight the serialisation might have on the request.

6.2.1.2 Energy Consumption

Table 10 - Question and metrics for energy consumption

Question	**Metric**
What is the difference in energy consumption of Protocol Buffers against JSON in HTTP-based REST architectures?	Average energy consumption (Joules)
	Median energy consumption (Joules)
	Max (Joules)
	Min (Joules)
	Energy consumption per request (Joules)

As previously stated, the primary focus of the dissertation is energy consumption. This subject is of increasing interest for the reasons already stated in Introduction 1. According to Table 10, two metrics can provide answers to the proposed questions. The first metric is the energy consumption measured in joules, and the second is the energy consumption per request in joules. The latter enables a more nuanced understanding of the weight of the serialisation method in the requests.

6.2.2 Performed Experiments

This section presents the results obtained from the experiments that were conducted, the environment, and the hypothesis tests that were performed.

It is worth noting that all the analyses performed here can be found in the reports generated by the tools used, these reports can be found inside each project in the k6 directory. It should also be noted that all tables or graphs were created manually using Python. These can be found in the "Data_Analysis" directory [48], which contains multiple Python files.

There are two types of Python files in that directory. The first type is the constants, which contain the paths and the extracted metric names. Each constant file is associated with the tests performed. For example, the file "constants_pynb_1000_gateway.py" contains constants that define the paths and metric names for the tests performed with 1,000 requests using the gateway.

The second file type is the Jupyter Notebook file, which contains the hypothesis tests performed and uses its associated constant file.

Finally, before experimenting, the payloads for the requests and responses were measured to verify if there is a causality effect between performance and energy efficiency, the size of the serialised or deserialised payloads. The values in question are located in Appendix E Table 28.

6.2.3 Performance Tests Setup

The execution of these tests was done using k6 [40], a tool that uses JavaScript to perform different types of performance tests, the possible tests have already been specified in the section 2.4.1. Furthermore, the tool also allows the generation of personalised metrics and is extensible with other packages using Webpack, which was needed for two of the requests.

Moreover, the tests performed were load tests, and it was defined that only a single virtual user (VU) would be used, with 1,000 and 10,000 requests, each iteration repeated 10 times. Table 11 shows all the tested endpoints for both JSON and Protocol Buffers serialisation methods. All the reports from the tests are available in the k6 directory, which contains not only the reports but also the scripts used.

Due to the high quantity of tests needed to be performed, a testing script was made in bash. This script, named “run_k6_tests.sh”, runs the k6 scripts and extracts energy consumption data from Prometheus and can be found in the script’s directory.

It should be noted that the analysis conducted in this section was based on a single report, namely the initial report generated. The underlying cause of this occurrence is that a single report has sample sizes of 1,000 and 10,000. The primary reason for conducting ten repetitions is to ensure the validity of the energy consumption analysis, which, if limited to a single repetition, would compromise the reliability of the study due to the insufficient sample size.

Table 11 - Endpoints requested in the tests

Title	Method	Endpoint	Description
Create user	POST	/users	Performs a POST request to create a user
Update user	PUT	/user/{id}	Performs a PUT request to update a user
Delete user	DELETE	/user/{id}	Performs a DELETE request to delete a user
Get all product entities	GET	/products/entities?orderby={id}&order={sort}	Performs a GET request to get all product entities
Get all users	GET	/users?orderby={id}&order={sort}	Performs a GET request to get all users in the system
Get user by ID	GET	/users/{id}	Performs a GET request to get a single user with a given ID

The ensuing results demonstrate that, under equivalent conditions, Protocol Buffers exhibits superior performance in the endpoints of user creation, all user retrieval, and all product entities retrieval, when compared with JSON. On the other hand, the investigation revealed that Protocol Buffers are slower than JSON in scenarios involving user retrieval by id, user update, and user deletion.

The following tables present the analysed data, it contains the executed test case, the number of requests performed in that same test case, the average response time in milliseconds, the throughput in requests per second, the median of the response time in milliseconds, the maximum response time registered in milliseconds and finally the minimum response time in milliseconds.

Table 12 - Benchmark results for creating users

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min (ms)
JSON	1,000	92.81	10.73	91.89	247.16	57.87
Protocol Buffers	1,000	89.84	11.05	89.50	243.52	58.38
JSON	10,000	90.61	10.99	92.25	273.92	58.96
Protocol Buffers	10,000	89.09	11.14	90.73	254.01	60.35
JSON + Gateway	1,000	92.30	10.79	90.90	255.46	59.21
Protocol Buffers + Gateway	1,000	89.94	11.04	89.01	280.27	67.65
JSON + Gateway	10,000	90.56	11.00	92.25	266.71	60.07
Protocol Buffers + Gateway	10,000	89.14	11.14	90.92	295.29	60.48

Table 12 shows that Protocol buffers were superior in every analysis performed for the creation of a user with 1,000 requests, and the absence of a gateway, Protocol Buffers exhibited a 3.20% lower average response time. Additionally, they demonstrated a 2.98% improvement in throughput in comparison to JSON and a 2.88% median reduction in response time compared to their counterpart. When increasing the load to 10,000 requests, the difference persisted, although not with the same magnitude of improvement, having the difference been slightly diminished.

It is noteworthy that the average response time with or without the gateway remained approximately constant, a result that is not observed in other endpoints, at least not across all test cases. The response times of the other endpoints varied by only a few milliseconds, with the gateway typically adding some overhead.

Table 13 - Benchmark results for retrieving all users

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min(ms)
JSON	1,000	5.75	164.31	2.75	72.50	1.80
Protocol Buffers	1,000	4.44	209.47	2.30	70.42	1.22
JSON	10,000	2.32	382.20	2.17	68.11	1.57
Protocol Buffers	10,000	1.99	430.92	2.10	40.49	1.56
JSON + Gateway	1,000	5.72	165.45	1.89	17.94	1.00
Protocol Buffers + Gateway	1,000	6.08	155.05	3.67	77.91	2.38
JSON + Gateway	10,000	3.52	259.99	3.39	14.92	2.32
Protocol Buffers + Gateway	10,000	3.24	276.02	3.24	12.37	1.89

Looking closely at Table 13, it shows that Protocol Buffers have superior performance in comparison to JSON under the same conditions in almost all endpoints. One explanation for this is due to their increased data requirement for serialisation, where Protocol Buffers gains a bigger margin of performance compared to the previously analysed data from Table 12. When performing 1,000 requests without a gateway, Protocol Buffers have a 22.78% reduction in average response time, they also have a 27.48% increase in throughput and a 16.36% reduction in median response time, showing a notable difference between the two serialisations.

Furthermore, in the absence of a gateway during the 10,000 requests, the discrepancy persists, resulting in Protocol Buffers exhibiting 14.22% reduction of average response time, 12.75% improved throughput, and 10% reduction in median response time.

Moreover, the only case where Protocol Buffers showed a worse performance was in the 1,000 requests with a gateway, showing an increased average response time of 6.29% in comparison to JSON. Nevertheless, the median shows the complete opposite with it being 5.18% more performant than JSON.

Table 14 - Benchmark results for retrieving user by id

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min(ms)
JSON	1,000	4.41	215.73	1.82	72.88	1.09
Protocol Buffers	1,000	4.17	227.40	1.80	44.34	1.03
JSON	10,000	1.49	590.14	1.36	62.64	0.84
Protocol Buffers	10,000	1.43	611.30	1.33	60.08	0.75
JSON + Gateway	1,000	5.02	189.54	3.19	18.70	0.68
Protocol Buffers + Gateway	1,000	4.74	199.75	3.04	64.10	1.90
JSON + Gateway	10,000	2.74	334.74	2.72	12.51	1.59
Protocol Buffers + Gateway	10,000	2.61	349.38	2.58	13.83	1.46

The Table 14 results show that when performing 1,000 requests without a gateway, shows that Protocol Buffers have a 5.44% improvement in the average response time in comparison to JSON, the throughput follows the same results, with a 5.41% reduction in the requests per second and finally, the median response time also has a 1.1% improvement, this performance improvement maintains itself in the other tests cases, with the differences in performance being around 4% to 5% on the average response time.

Table 15 - Benchmark results for retrieving all product entities

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min(ms)
JSON	1,000	19.27	51.18	5.36	589.20	2.75
Protocol Buffers	1,000	13.67	71.79	4.42	510.97	2.30
JSON	10,000	2.02	447.96	1.85	60.55	0.95
Protocol Buffers	10,000	1.86	479.79	1.73	41.07	0.89
JSON + Gateway	1,000	20.71	47.65	9.61	641.16	4.68
Protocol Buffers + Gateway	1,000	15.60	62.95	6.11	545.20	3.40
JSON + Gateway	10,000	3.35	277.32	3.32	57.12	1.74
Protocol Buffers + Gateway	10,000	3.15	292.83	3.08	58.45	1.54

Table 15 shows that Protocol Buffers are more performant than JSON, the retrieval of all product entities is one of the more serialisation-heavy requests, and it shows that Protocol Buffers prosper more in these kinds of requests with high data density.

Furthermore, when analysing the data, the 1,000 requests without a gateway, the tests show a 29.06% average response time reduction on Protocol Buffers, with an increase in throughput of 40.27% and a decrease in the median response time of 17.54%. Moreover, when increasing the number of requests performed, the difference in performance diminishes a lot, in this case, making the average response time difference decrease about 7.92%, increasing the throughput by 7.10% and decreasing the median response time by 6.49%.

Finally, when analysing the tests with a gateway, we can reach the same conclusion that Protocol Buffers are more performant than JSON in this endpoint. In the 1,000 requests, it registered a 36.42% decrease in terms of the median response time and a 24.67% average response time reduction, and the throughput is also 32.11% higher than its counterpart. Even in the 10,000 requests tests with a gateway, the same pattern can be observed, even if it diminished quite a lot.

Table 16 - Benchmark results for updating a user

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min(ms)
JSON	1,000	21.35	46.14	15.71	91.34	13.06
Protocol Buffers	1,000	21.91	44.60	16.90	312.73	11.97
JSON	10,000	18.90	52.04	15.01	212.5	11.97
Protocol Buffers	10,000	18.37	53.02	15.28	195.6	12.02
JSON + Gateway	1,000	22.11	44.58	17.18	119.59	14.33
Protocol Buffers + Gateway	1,000	21.07	46.36	17.41	112.97	14.43
JSON + Gateway	10,000	19.74	49.87	16.75	235.24	13.53
Protocol Buffers + Gateway	10,000	19.62	49.73	17.25	224.08	13.60

Table 16 shows the data from the user update performance tests, which is one of the least data-intensive endpoints being tested. The first test case, the 1,000 requests without a gateway, shows a 2.62% increase in average response time on Protocol Buffers, and the median also increased by 7.57% more in comparison to JSON, with the throughput also being lower by 3.34% in comparison to JSON

Going forward, the rest of the tests showed that Protocol Buffers have slightly better performance than JSON in terms of average response times, but looking closely at the throughput, Protocol Buffers were superior in the 10,000 requests without a gateway and the 1,000 requests with a gateway, with the biggest difference being a 3.99% increase in comparison to JSON.

On the other hand, looking at the median response time, it shows that Protocol Buffers were inferior in every test performed, which is contradictory to what the throughput shows. Nonetheless, it is important to note that these differences, even though small, should still be considered.

Table 17 - Benchmark results for deleting a user

Metric / Test Case	Requests	Average Response Time (ms)	Throughput (req/s)	Median(ms)	Max(ms)	Min(ms)
JSON	1,000	19.64	50.35	14.61	85.08	12.00
Protocol Buffers	1,000	20.06	49.31	15.12	200.86	12.11
JSON	10,000	16.42	60.08	13.04	57.27	10.39
Protocol Buffers	10,000	15.97	61.77	13.15	73.14	10.30
JSON + Gateway	1,000	20.33	48.65	16.30	89.46	13.43
Protocol Buffers + Gateway	1,000	21.59	45.84	16.89	298.03	13.77
JSON + Gateway	10,000	16.58	59.51	14.39	76.64	11.70
Protocol Buffers + Gateway	10,000	16.88	58.47	14.89	48.99	11.72

Table 17 shows the data of the deletion of a user, representing the endpoint with the least required serialisation to be performed. Starting with the first test case, which was executed with 1,000 requests without a gateway. The data analysis revealed that JSON outperforms Protocol Buffers in terms of average response time, increasing by 2.14%, there is also a 2.49% increase in the median response time and a 2.06% decrease in throughput.

Furthermore, when performing 10,000 requests without a gateway, the metrics change in favour of Protocol Buffers, with a 2.74% reduction in average response time, also there is a 2.81% increase in throughput. Nevertheless, the median response time showed a 0.84% increase in the same test, which is contradictory to the other analysed metrics.

Finally, when analysing both tests where the gateway was added, it is possible to see that in the 1,000 requests test case, Protocol Buffers were inferior, registering an average response time increase of 6.2%, a median increase of 3.62%, resulting in an 5.77% reduction in throughput.

However, when performing 10,000 requests with a gateway, this difference in performance diminishes quite significantly, having an 1.81% increase in average response time, the an increase of 3.47% for the median, and an increase of 1.71% in throughput.

6.2.3.1 Hypothesis Tests

In order to further validate the obtained analysis, hypothesis tests were performed. These are statistical methods used to validate our hypothesis. For this task, Python was used with the SciPy [59] library, which offers all the algorithms for statistical analysis. In essence, these tests serve to verify if the performance differences between JSON and Protocol Buffers are statistically significant.

The hypotheses were developed for each endpoint because each endpoint has its own logic and complexity, and aggregating all the results would mask the differences, which could lead to misleading conclusions.

The following tests followed a simple procedure to ensure an adequate statistical method was chosen. Firstly, each dataset was tested for normality, which means each dataset was analysed to see if the data followed a normal distribution.

As stated before, our dataset for each endpoint exceeds at least 1,000 requests, which means that a low-number requirement test wasn't possible, like the Shapiro-Wilk [60], as such, a test that doesn't view the small deviations is important, the one chosen was D'Agostino and Pearson's [61], for two main reasons, the first one being that the SciPy library uses that one as the default for the normality test [62], but the other reason and a more important one is because of the dataset size.

Finally, after uncovering the normality of the dataset, we choose the test. If the distribution is normal, we use a parametric test, in this case, the T-test [63], however, if it doesn't follow a normal distribution, a non-parametric test needs to be employed, in this case, it was used the Mann-Whitney U [64], which works well with large datasets. Table 18 shows the created hypothesis for the performance study.

Table 18 - Hypothesis tests for each endpoint in terms of performance

Title	**Method**	**Hypothesis**
Create user	POST	Protocol Buffers have a lower response time than JSON
Get all product entities	GET	
Get all users	GET	
Get user by Id	GET	
Update user	PUT	
Delete user	DELETE	

```
json_norm_p = normaltest(json_data).pvalue
proto_norm_p = normaltest(protobuf_data).pvalue
var_p = levene(protobuf_data, json_data).pvalue

if json_norm_p > 0.05 and proto_norm_p > 0.05:
  _, p_val = ttest_ind(protobuf_data, json_data, equal_var=(var_p >= 0.05))
  test_name = "t-test"
else:
  _, p_val = mannwhitneyu(protobuf_data, json_data, alternative="less")
  test_name = "Mann-Whitney U"
return {
  "p_value": p_val,
  "test_used": test_name,
  "json_median": np.median(json_data),
  "protobuf_median": np.median(protobuf_data),
  "is_significant": (
     "The difference is significant"
     if p_val < 0.05
     else "The difference is not significant"
   ),
}
```

Code Snippet 17 - Response time hypothesis test structure

Code Snippet 17 is the base structure to perform these tests. The code and the results can all be found in the “Data_Analysis” directory.

The results of the hypothesis tests for the response time are as follows. The tests have shown statistical significance in the user creation process across all performed tests, both with and without the gateway, aligning with the findings of prior analyses that consistently demonstrated the superiority of Protocol Buffers. Consequently, the alternative hypothesis (H1) is substantiated by the results of all the tests conducted in this endpoint.

The retrieval of all users exhibited a similar phenomenon, demonstrating that there is statistical significance across all the tests performed. This is consistent with the prior analysis performed, not only that, but this is one of the most intense serialisation endpoints, where Protocol Buffers have been shown to outperform JSON. Like the previous endpoint, this one also accepts the alternative hypothesis throughout all the tests performed in this endpoint.

The next endpoint, however, is slightly different, the retrieval of user information by its id doesn’t show statistical significance in one out of the four tests. The single instance where statistical significance is absent is observed in the 1,000 requests without a gateway, accepting the null hypothesis (H0).

On the other hand, the other three tests, in this case, the 10,000 requests with and without a gateway and the 1,000 requests with a gateway, have shown to be statistically significant, accepting the alternative hypothesis, which states that Protocol Buffers are superior in terms of performance than JSON.

The retrieval of every product entity endpoint is no different from the first two endpoints analysed, as it demonstrates that, in all tests, there is a significant statistical difference. This

finding is also consistent with the analysis carried out, and it's also a serialisation-heavy endpoint, where Protocol Buffers have been observed to have superior overall performance. With everything said, the results of all tests conducted at this endpoint support the alternative hypothesis.

The endpoint for updating user information demonstrates that there is no statistical difference in all the tests performed, as Protocol Buffer never demonstrated superior performance in comparison to JSON in this case, except in the average response time of the last test case when performing 10,000 requests with a gateway, but even there, the difference was marginal. This is also one of the endpoints for which the amount of data required for serialisation is minimal. In addition, the null hypothesis was confirmed in all instances following the execution of the designated tests at this endpoint.

Finally, the last endpoint, the deletion of a user, has three out of four tests that are not statistically significant, where the only one that was significant was the test with 10,000 requests without a gateway. This is the endpoint with the least serialisation out of the six endpoints tested. Like the user information update.

6.2.4 Energy Consumption Tests

To perform these tests, Kepler was used, which is a tool that can extract the energy consumption of a software system inside a Kubernetes cluster, for further details, check section 2.4. Furthermore, Kepler is a Prometheus exporter, with this, we can connect with Grafana and use their pre-made dashboard [65], after that, we change the scrape interval to one second, meaning that each second, Grafana queries Prometheus for Kepler metrics.

To further explain how the tests proceeded, during the creation of the k6 performance tests, it was created, for each script, a way to see the time the test took, giving the start and the end time of the tests, this gives the exact time frame to export the energy consumption of the POD during tests, allowing for a precise and consistent way to get the necessary data from Kepler.

It should be noted that, as outlined in section 6.2.3, the analysis conducted in this section utilised the 10 generated reports to calculate mean values and to enable the execution of some form of hypothesis tests.

For this section, an analysis of the energy consumption data is going to be made, which comprises a series of tables, with each table corresponding to a separate endpoint that has been tested. The tables provide test cases, indicating the utilisation of either JSON or Protocol Buffers, with or without a gateway.

The tables further enumerate the number of requests executed for that test, the average and median energy consumption in joules, and finally, the energy consumption per request, which is calculated through the division of the average energy consumption by the number of requests. This is possible because the average energy consumption is a total average of the energy consumed per repetition, so it is an average of the total energy consumed by the 10 reports.

Table 19 - Energy consumption of user creation in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(J)	Energy consumption per request (J)
JSON	1,000	357.34	355.49	376.66	347.80	0.36
Protocol Buffers	1,000	360.65	361.20	372.82	347.88	0.36
JSON	10,000	4300.34	4304.18	4801.95	3672.11	0.43
Protocol Buffers	10,000	4361.49	4339.10	5082.16	3694.91	0.44
JSON + Gateway	1,000	378.33	374.33	399.14	369.33	0.38
Protocol Buffers + Gateway	1,000	379.03	378.17	388.72	375.56	0.38
JSON + Gateway	10,000	4525.71	4582.42	5026.46	3865.06	0.45
Protocol Buffers + Gateway	10,000	4538.43	4577.74	5111.56	3844.18	0.45

Table 19 shows the gathered energy data for creating a user endpoint. Upon analysing that data, it clearly shows that Protocol Buffers, on average, consumed more energy than JSON in every test, even though the differences are marginal. However, the most significant registered disparity was observed in the test involving 10,000 requests without a gateway, where Protocol Buffers' average energy consumption was 1.42% higher than that of JSON.

It should be noted that the implementation of a gateway has been observed to increase energy consumption. In this instance, the observed differences were not pronounced. However, it is important to note that the results of subsequent tests may vary.

Table 20 - Energy consumption of the retrieval of all users in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(J)	Energy consumption per request (J)
JSON	1,000	9.82	8.72	26.45	0.49	0.01
Protocol Buffers	1,000	5.03	3.67	15.10	0.01	<0.01
JSON	10,000	148.21	146.55	160.62	139.19	0.01
Protocol Buffers	10,000	86.19	85.25	89.40	81.83	0.01
JSON + Gateway	1,000	22.59	24.82	43.87	8.55	0.02
Protocol Buffers + Gateway	1,000	17.05	17.01	36.93	5.45	0.02
JSON + Gateway	10,000	299.08	296.85	314.13	288.85	0.03
Protocol Buffers + Gateway	10,000	233.97	235.49	240.92	223.41	0.02

As illustrated in Table 20, the data about the energy consumption of the retrieval process for all users is presented in joules. The gathered data exhibits that Protocol Buffers were consistently better than JSON by a great margin, with the lowest difference being a 21.77% less

energy consumption on average from Protocol Buffers in comparison to JSON, the test where this happened was the one with 10,000 requests with a gateway.

It is important to note that a close examination of the data suggests a clear implication regarding the utilisation of a gateway. When not in use, the percentage differences in terms of average energy consumption are observed to exceed 40%. However, when the gateway is employed, these differences are reduced to approximately half, resulting in an average energy consumption percentage difference that ranges from 20% to 24%.

Table 21 - Energy consumption of the retrieval of a user by its id in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(J)	Energy consumption per request (J)
JSON	1,000	4.09	1.54	20.9	0.01	<0.01
Protocol Buffers	1,000	2.72	1.00	11.09	0.00	<0.01
JSON	10,000	45.77	43.83	52.51	42.66	<0.01
Protocol Buffers	10,000	44.78	44.65	46.90	43.56	<0.01
JSON + Gateway	1,000	15.19	16.14	34.00	40.9	0.01
Protocol Buffers + Gateway	1,000	11.02	9.31	24.80	0.08	0.01
JSON + Gateway	10,000	176.57	175.52	185.87	165.87	0.02
Protocol Buffers + Gateway	10,000	167.19	166.36	173.32	159.02	0.02

Table 21 shows the gathered data for the energy consumption of the retrieval of a user by its id. This test doesn't differ too much in the results in comparison to the previously analysed endpoint, at least in which is superior, in this case, Protocol Buffers is better in every test. When looking at the tests with 1,000 requests with and without a gateway, these differences are huge, with the test without the gateway having a 33.50% decrease in energy consumption on average, and the test with the gateway showing a 27.45% reduction.

On the other hand, upon analysing the tests with 10,000 requests, the percentage difference reduces greatly, with the test without a gateway resulting in a 2.16% reduction and the test with the gateway resulting in a 5.31% difference. It is important to add that, as has been previously demonstrated, the incorporation of a gateway has been shown to result in a substantial increase in energy consumption.

Table 22 - Energy consumption of the retrieval of all product entities in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(J)	Energy consumption per request (J)
JSON	1,000	22.91	15.23	97.59	4.42	0.02
Protocol Buffers	1,000	18.66	18.03	58.89	3.64	0.02
JSON	10,000	80.19	81.05	84.59	73.85	<0.01
Protocol Buffers	10,000	73.12	72.92	80.28	69.49	<0.01
JSON + Gateway	1,000	51.29	40.06	178.66	8.19	0.05
Protocol Buffers + Gateway	1,000	31.78	28.89	72.88	5.61	0.03
JSON + Gateway	10,000	216.30	215.75	232.72	203.72	0.02
Protocol Buffers + Gateway	10,000	204.99	208.63	211.87	211.87	0.02

As illustrated in Table 22, the collected data pertains to the energy consumption of the retrieval of all product entities. As expected, it shows that protocol buffers are more energy efficient in comparison to JSON across all tests performed.

Although Protocol Buffers are superior, there is a clear loss in efficiency when comparing 1,000 requests with 10,000 requests, with or without a gateway, for example, in the test with 1,000 requests without a gateway the difference in the average energy consumption is 18.55% but in the 10,000 requests without a gateway the difference is only 8.82%, this is a reoccurring pattern across all tests involving data retrieval.

Furthermore, one detail is the median energy consumption, which, looking closely at the test with 1,000 requests without a gateway, shows that the median in JSON is better than Protocol Buffers by 18.38%.

Table 23 - Energy consumption of updating a user in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(J)	Energy consumption per request (J)
JSON	1,000	73.30	73.15	79.58	67.17	0.07
Protocol Buffers	1,000	76.30	76.40	79.88	73.37	0.08
JSON	10,000	763.24	760.76	775.94	753.38	0.08
Protocol Buffers	10,000	812.03	814.23	829.51	786.99	0.08
JSON + Gateway	1,000	88.89	87.22	101.25	79.57	0.09
Protocol Buffers + Gateway	1,000	89.15	88.83	97.82	82.02	0.09
JSON + Gateway	10,000	912.17	910.89	931.96	895.57	0.09
Protocol Buffers + Gateway	10,000	945.42	943.42	964.31	925.31	0.09

Table 23 shows the energy consumption of the test to update the user information. This evaluation constitutes a significant challenge for Protocol Buffers, as all test cases in this endpoint reveal that Protocol Buffers have increased energy consumption compared to JSON. The biggest difference is in the test with 10,000 requests without a gateway, showing a 6.39% reduced energy consumption on average in comparison to JSON.

Table 24 - Energy consumption of deleting a user in joules

Metric / Test Case	Requests	Average Energy Consumption (J)	Median Energy Consumption (J)	Max(J)	Min(j)	Energy consumption per request (J)
JSON	1,000	66.71	65.31	74.98	69.62	0.07
Protocol Buffers	1,000	66.65	65.29	80.85	57.30	0.07
JSON	10,000	364.71	345.71	660.38	85.83	0.04
Protocol Buffers	10,000	376.93	355.23	664.42	96.86	0.04
JSON + Gateway	1,000	80.06	80.40	89.56	72.03	0.08
Protocol Buffers + Gateway	1,000	83.07	84.11	93.03	74.64	0.08
JSON + Gateway	10,000	525.59	518.80	804.02	248.75	0.05
Protocol Buffers + Gateway	10,000	537.43	525.56	808.43	263.54	0.05

Finally, Table 24 shows the energy consumption of the endpoint to delete a user. The initial test case, involving 1,000 requests without a gateway, revealed that Protocol Buffers were just marginally better, just expending 0.09% less than JSON. However, throughout all the other tests, Protocol Buffers show around a 2% to 3% increase in energy consumption on average, which is expected, as this was the least serialisation-intensive endpoint and Protocol Buffers might add unnecessary overhead for such a small response payload.

Finally, throughout this section, it is clear that the gateway made the energy consumption increase in comparison to its homologous test without a gateway, in some cases, even more than doubling the consumption, probably making this one of the biggest deterrents to its adoption, to maintain the documentation of both systems.

6.2.4.1 Hypothesis Tests

It is unfortunate that, due to the limited amount of data available, the use of traditional parametric and non-parametric hypothesis tests is not feasible. The statistical power of these tests is negligible, and they are susceptible to errors, which may result in incorrect analysis.

In addition, there are several methodologies for conducting statistical analysis with a limited sample size. The most effective approach involves the implementation of a paired bootstrap technique [66], which generates confidence intervals [67]. Consequently, this process can also yield a p-value, allowing to do hypothesis tests.

Table 25 - Hypothesis tests for each endpoint in terms of energy consumption

Title	Method	Hypothesis
Create user	POST	Protocol Buffers are more energy efficient than JSON
Get all product entities	GET	
Get all users	GET	
Get user by Id	GET	
Update user	PUT	
Delete user	DELETE	

```
for test_num, test_data in data_dict.items():
  for size, results in test_data.items():
    plot_ecdf(
    [result.values for result in results],
    f"Test {results[0].name} Size {size}",
      size
     )

    d = (results[1].values, results[0].values,)
    bootstrap_res = bootstrap(
      d,
      statistic=lambda x, y: np.mean(x - y),
      paired=True,
      vectorized=False,
      n_resamples=10000,
    )
    ci_low, ci_high = bootstrap_res.confidence_interval
    print(f"Bootstrap 95% CI for difference: [{ci_low:.2f},
{ci_high:.2f}]")

    bootstrap_diffs = bootstrap_res.bootstrap_distribution
    prob_B_gt_A = np.mean(bootstrap_diffs < 0)
    p_value = np.mean(bootstrap_diffs >= 0)
    print(
      f"{results[0].name} - Probability Protocol Buffers < JSON:
{prob_B_gt_A:.1%}"
    )
    print(f"Bootstrap p-value: {p_value:.4f}")
    print(f"Conclusion: {'Reject H0' if p_value < 0.05 else 'Fail to reject
H0'}\n")
```

Code Snippet 18 - Structure for the energy consumption bootstrap hypothesis test

Code Snippet 18 demonstrates the implementation of the bootstrap technique, utilising the SciPy library. In essence, the bootstrap method is employed to do 10,000 resamples, which consequently yield the confidence intervals and the distribution of the data. Through the distribution, the p-value for the hypothesis test is calculated.

All the analyses performed for this section can be found in the repository [48] in a Jupyter notebook file named “energy_consumption_data.ipynb”.

The results of the hypothesis tests for the energy consumption are as follows. The hypothesis tests for the user creation endpoint data demonstrate that it fails to reject the null hypothesis across all tests performed, including those conducted with the gateway.

Furthermore, for the test to retrieve all users' data, the hypothesis tests demonstrated that the test consistently rejected the null hypothesis across all tests performed, thereby evidencing a clear superiority in this heavy data serialisation endpoint.

Moreover, the hypothesis tests employed for the retrieval of user data by their id are somewhat divergent. In three out of four tests, the null hypothesis was not rejected. The only test in which the null hypothesis was rejected was one in which 10,000 requests were made via a gateway.

For the endpoint to retrieve all product entities, the null hypothesis is rejected in three out of four tests. The sole test in which the null hypothesis is not rejected is the one in which 1,000 requests are made without a gateway.

Finally, for the final two endpoints, the hypothesis tests were unsuccessful in rejecting the null hypothesis across all tests conducted.

6.3 Conclusion

The subsequent section offers a conclusion regarding the capabilities that Protocol Buffers can offer in comparison to those of JSON within the same HTTP REST architecture.

Initially, with regard to performance, Protocol Buffers have demonstrated clear superiority, in terms of performance, over JSON in instances where high-density serialisation is necessary, such as the creation of a user, the retrieval of all users and the retrieval of product entities. It is possible that in some other specific cases, Protocol Buffers also exhibit better performance. For instance, in the context of deleting a user, the execution of 10,000 requests without a gateway was found to be faster when using Protocol Buffers.

Secondly, with respect to energy efficiency, through a statistical analysis, Protocol Buffers are far superior in the retrieval endpoints, on the other hand, endpoints where the delete, put and post are done, Protocol Buffers fall of, this could be due to the lower necessity of serialisation, and because of that added overhead, Protocol Buffers waste more energy.

The analysis indicates that Protocol Buffers demonstrate superior performance in scenarios involving high serialisation intensity when compared to JSON, as previously analysed in the literature review. With regard to energy consumption, it was observed that the analysis followed the same pattern as the performance, indicating that when the scenario involves high serialisation, its energy consumption seems to be lower.

An important conclusion is that adding an API gateway has hindered the system's performance and energy consumption. While the difference in performance is less notable, the difference in energy consumption is significant, at almost double the amount, making it questionable to have an API gateway.

For a more detailed view of the processes that were run, refer to the repository [48] where all the data, documents, setup and project can be found.

7 Conclusion

The final chapter contains the accomplishments throughout the dissertation's development process, showcasing the outcomes of the conducted analyses and the insights derived from the study.

Moreover, it delineates the difficulties encountered related to personal knowledge and implementation. The chapter concludes with a discussion of threats to validity that could be posed to the study performed.

7.1 Accomplishments

The study successfully conducted a comparative analysis, thereby providing insights into the discrepancy in performance and energy efficiency between Protocol Buffers and JSON when applied within the same REST architecture. This achievement was made possible by the comprehensive testing and subsequent analysis that exposed the strengths and limitations of Protocol Buffers when employed within the same REST architecture as JSON. A notable accomplishment of this study is the knowledge it has generated, encompassing not only concepts and statistics but also the technologies employed.

Another salient topic was the results obtained from the Grafana k6 reports and Kepler, which elucidate the circumstances under which Protocol Buffers might be a preferable serialisation format, and which might be better or worse in terms of energy efficiency or performance. As anticipated, as the volume of data to be serialised increases, the performance and energy efficiency of Protocol Buffers compared to JSON improve.

7.2 Difficulties

The development of the dissertation was accompanied by numerous challenges, including issues with the tooling. Some tools functioned improperly, while others were not compatible with the required tasks. For instance, Kepler, a novel tool, exhibited some deficiencies, which is not unexpected for a new tool. In this case, the primary issue was the inability of the tool to gather energy consumption data on a Windows machine, rendering the initial tests futile. This led to significant confusion concerning the adequacy of the tool and the validity of the research methods.

However, upon transitioning to a Linux operating system, the tool demonstrated the capacity to gather the necessary data, thereby resolving the issue. A further complication, related to Grafana k6, emerged when it became apparent that the tool had not been designed to transmit Protocol Buffers via HTTP in the manner of JSON. This required a significant degree of juggling, namely the integration of Webpack to allow the utilisation of the Protocol Buffers NPM package, which was essential for the execution of tests with Protocol Buffers.

Another challenge that was encountered was during the integration process of the Protocol Buffers, a serialisation format with which I had no prior experience. The complexity of debugging these tools contributed to the overall complexity of the implementation process.

7.3 Threats to Validity

The validity of the results obtained in the study may be called into question by some factors. One such factor pertains to the selection of the project, with subsequent migration being another salient factor. From the start, there was a huge effort to ensure that the choices made during the dissertation development process were not influenced by personal bias.

However, it is acknowledged that, even if these choices were not made with a strong sense of personal preference, they may still carry an implicit element of partiality. This partiality can be exemplified by the application of best practices during the migration process, the maintenance of code logic, or the selection of a tool for a specific task.

A further and final issue is the inexperience with Protocol Buffers, which can result in difficulties in comprehending the scope of acceptable practices. This challenge can be overcome by individuals with more experience with the technology, allowing them to discern potential issues in the work being conducted.

References

[1] Green Software Foundation, “Energy Efficiency | Learn Green Software.” Accessed: Jun. 02, 2025. [Online]. Available: https://learn.greensoftware.foundation/energy-efficiency/

[2] Green Software Foundation, “Reduce transmitted data | Green Software Patterns.” Accessed: Jun. 02, 2025. [Online]. Available: https://patterns.greensoftware.foundation/catalog/cloud/reduce-transmitted-data/

[3] M. Lampert, “Trend Report: Global Rise in Environmental Concern.” Accessed: Jun. 02, 2025. [Online]. Available: https://glocalities.com/reports/environmental-concern

[4] A. Sumaray and S. K. Makki, “A comparison of data serialization formats for optimal efficiency on a mobile platform,” *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, ICUIMC’12*, 2012, doi: 10.1145/2184751.2184810.

[5] J. C. Viotti and M. Kinderkhedia, “A Benchmark of JSON-compatible Binary Serialization Specifications,” Jan. 2022, Accessed: Nov. 02, 2024. [Online]. Available: https://arxiv.org/abs/2201.03051v1

[6] Apache Software Foundation, “Apache Avro.” Accessed: Jun. 11, 2025. [Online]. Available: https://avro.apache.org/

[7] S. Furuhashi, “MessagePack: It’s like JSON. but fast and small.” Accessed: Jun. 11, 2025. [Online]. Available: https://msgpack.org/

[8] Apache Software Foundation, “Apache Thrift - Home.” Accessed: Jun. 11, 2025. [Online]. Available: https://thrift.apache.org/

[9] Google LLC, “Overview | Protocol Buffers Documentation.” Accessed: Nov. 28, 2024. [Online]. Available: https://protobuf.dev/overview/

[10] T. Myastovskiy, “Evaluating the Performance of Serialization Protocols in Apache Kafka,” 2024, Accessed: Nov. 02, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-227363

[11] B. Gil and P. Trezentos, “Impacts of data interchange formats on energy consumption and performance in smartphones,” *ACM International Conference Proceeding Series*, vol. 1, pp. 1–6, 2011, doi: 10.1145/2016716.2016718.

[12] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, “Performance evaluation of using Protocol Buffers in the Internet of Things communication: Protobuf vs. JSON/BSON comparison with a focus on transportation’s IoT,” *Proceedings of 2016 International*

Conference on Smart Systems and Technologies, SST 2016, pp. 261–265, Dec. 2016, doi: 10.1109/SST.2016.7765670.

[13] D. P. Proos and N. Carlsson, “Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV,” in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 10–18.

[14] A. Neumann, N. Laranjeiro, and J. Bernardino, “An Analysis of Public REST Web Service APIs,” *IEEE Trans Serv Comput*, vol. 14, no. 4, pp. 957–970, Jul. 2021, doi: 10.1109/TSC.2018.2847344.

[15] S. Hao and B. Yu, “The Impact of Technology Selection on Innovation Success and Organizational Performance,” *iBusiness*, vol. 03, no. 04, pp. 366–371, 2011, doi: 10.4236/IB.2011.34049.

[16] M. Newton Hedelin, “Benchmarking and performance analysis of communication protocols : A comparative case study of gRPC, REST, and SOAP,” 2024, *KTH Royal Institute of Technology*. Accessed: Nov. 02, 2024. [Online]. Available: https://kth.diva-portal.org/smash/get/diva2:1887929/FULLTEXT01.pdf

[17] M. Johansson and O. Isabella, “Comparative Study of REST and gRPC for Microservices in Established Software Architectures,” 2023. Accessed: Nov. 02, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-195563

[18] C. Joumaa and S. Kadry, “Green IT: Case Studies,” *Energy Procedia*, vol. 16, no. PART B, pp. 1052–1058, Jan. 2012, doi: 10.1016/J.EGYPRO.2012.01.168.

[19] M. Woodside, G. Franks, and D. C. Petriu, “The future of software performance engineering,” *FoSE 2007: Future of Software Engineering*, pp. 171–187, 2007, doi: 10.1109/FOSE.2007.32.

[20] G. Procaccianti, H. Fernández, and P. Lago, “The Journal of Systems and Software Empirical evaluation of two best practices for energy-efficient software development,” *J Syst Softw*, vol. 117, pp. 185–198, 2016, doi: 10.1016/j.jss.2016.02.035.

[21] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, “What Do Programmers Know about Software Energy Consumption?,” *IEEE Softw*, vol. 33, no. 3, pp. 83–89, May 2016, doi: 10.1109/MS.2015.83.

[22] Green Software Foundation, “Green Software Foundation | GSF.” Accessed: Jun. 02, 2025. [Online]. Available: https://greensoftware.foundation/

[23] A. J. Ko, T. D. LaToza, and M. M. Burnett, “A practical guide to controlled experiments of software engineering tools with human participants,” *Empir Softw Eng*, vol. 20, no. 1, pp. 110–141, Feb. 2013, doi: 10.1007/S10664-013-9279-3.

[24] P. Bhandari, “What Is a Controlled Experiment? | Definitions & Examples.” Accessed: Dec. 25, 2024. [Online]. Available: https://www.scribbr.com/methodology/controlled-experiment/

[25] D. I. K. Sjøberg *et al.*, “A survey of controlled experiments in software engineering,” *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 733–753, Sep. 2005, doi: 10.1109/TSE.2005.97.

[26] IEEE Board of Directors, “IEEE Code of Ethics | IEEE.” Accessed: May 23, 2025. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8

[27] Ordem dos Engenheiros, “Código de Ética e Deontologia”, Accessed: Jun. 25, 2025. [Online]. Available: https://www.ordemdosengenheiros.pt/fotos/gca/blocks_items/codigo_ed_143278024064df56810f53a.pdf

[28] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Doctoral dissertation, University of California, Irvine, 2000. Accessed: Jun. 25, 2025. [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[29] AltexSoft Editorial Team, “What is a REST API? Beginner’s Guide.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.altexsoft.com/blog/rest-api-design/

[30] L. Gupta, “REST Architectural Constraints.” Accessed: Nov. 28, 2024. [Online]. Available: https://restfulapi.net/rest-architectural-constraints/

[31] JSON.org, “JSON.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.json.org/json-en.html

[32] V. Buono, P. Petrovic Author Vincenzo Buono Petar Petrovic, and S. Fredrik Stridh Examiner, “Enhance Inter-service Communication in Supersonic K-Native REST-based Java Microservice Architectures,” 2021, Accessed: Nov. 02, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:hkr:diva-22135

[33] P. Mooney and M. Minghini, “GEOSPATIAL DATA EXCHANGE USING BINARY DATA SERIALIZATION APPROACHES,” *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLVIII-4-W1-2022, no. 4/W1-2022, pp. 307–313, Aug. 2022, doi: 10.5194/ISPRS-ARCHIVES-XLVIII-4-W1-2022-307-2022.

[34] C. Currier, “Protocol Buffers,” in *Mobile Forensics – The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*, C. Hummert and D. Pawlaszczyk, Eds., Cham: Springer International Publishing, 2022, pp. 223–260. doi: 10.1007/978-3-030-98467-0_9.

[35] J. Berg and D. Mebrahtu Redi, “Benchmarking the request throughput of conventional API calls and gRPC : A Comparative Study of REST and gRPC,” vol. TRITA-EECS-EX, 2023, Accessed: Nov. 02, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-334990

[36] G. Kaur and M. M. Fuad, “An evaluation of protocol buffer,” *Conference Proceedings - IEEE SOUTHEASTCON*, pp. 459–462, 2010, doi: 10.1109/SECON.2010.5453828.

[37] G. Bath, R. Black, A. Podelko, A. Pollner, and R. Rice, “Certified Tester Foundation Level Syllabus-Performance Testing,” 2018, Accessed: Dec. 25, 2024. [Online]. Available: https://astqb.org/assets/documents/ISTQB-CTFL-PT-Syllabus-2018.pdf

[38] S. Pargaonkar, “A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering,” *International Journal of Science and Research (IJSR)*, vol. 12, no. 8, pp. 2008–2014, Aug. 2023, doi: 10.21275/SR23822111402.

[39] Apache Software Foundation, “Apache JMeter - Apache JMeter™.” Accessed: Dec. 25, 2024. [Online]. Available: https://jmeter.apache.org/

[40] Grafana Team, “Grafana k6 | Grafana k6 documentation.” Accessed: Mar. 17, 2025. [Online]. Available: https://grafana.com/docs/k6/latest/

[41] D. Wang, “Meeting green computing challenges,” *10th Electronics Packaging Technology Conference, EPTC 2008*, pp. 121–126, 2008, doi: 10.1109/EPTC.2008.4763421.

[42] PowerAPI team, “PowerAPI.” Accessed: May 26, 2025. [Online]. Available: https://powerapi.org/

[43] Kepler’s team, “Deploy using Manifests - Kepler.” Accessed: Dec. 26, 2024. [Online]. Available: https://sustainable-computing.io/installation/kepler/

[44] M. Amaral *et al.*, “Process-Based Efficient Power Level Exporter,” *IEEE International Conference on Cloud Computing, CLOUD*, pp. 456–467, 2024, doi: 10.1109/CLOUD62652.2024.00058.

[45] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *BMJ*, vol. 372, Mar. 2021, doi: 10.1136/BMJ.N71.

[46] “PRISMA 2020 flow diagram — PRISMA statement.” Accessed: Dec. 04, 2024. [Online]. Available: https://www.prisma-statement.org/prisma-2020-flow-diagram

[47] E. Maltsev and O. Muliarevych, “BEYOND JSON: EVALUATING SERIALIZATION FORMATS FOR SPACE-EFFICIENT COMMUNICATION,” *ADVANCES IN CYBER-PHYSICAL SYSTEMS*, vol. 9, no. 1, 2024, doi: 10.23939/acps2024.01.009.

[48] M. Ferreira, “MiguelFerreira18/Comparing_JSON_and_ProtoBuf_in_HTTP-based-_REST_architectures.” Accessed: Mar. 25, 2025. [Online]. Available: https://github.com/MiguelFerreira18/Comparing_JSON_and_ProtoBuf_in_HTTP-based-_REST_architectures

[49] Project Management Institute, “A guide to the project management body of knowledge / Project Management Institute (PMBOK),” *Project Management Institute, Inc.*, p. 762, 2017.

[50] D. Rezende, “diegorezm/convenience.store.api.” Accessed: Mar. 11, 2025. [Online]. Available: https://github.com/diegorezm/convenience.store.api/tree/main

[51] SIG Minikube, “minikube start | minikube.” Accessed: Mar. 13, 2025. [Online]. Available: https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download

[52] Kepler’s team, “Deploy using Helm Chart - Kepler.” Accessed: Mar. 13, 2025. [Online]. Available: https://sustainable-computing.io/installation/kepler-helm/

[53] Nodejs Team, “Node.js — Run JavaScript Everywhere.” Accessed: Jun. 10, 2025. [Online]. Available: https://nodejs.org/en

[54] “OpenAPI Initiative – The OpenAPI Initiative provides an open source, technical community, within which industry participants may easily contribute to building a vendor-neutral, portable and an open specification for providing technical metadata for REST APIs – the ‘OpenAPI Specification’ (OAS).” Accessed: May 28, 2025. [Online]. Available: https://www.openapis.org/

[55] M. Ferreira, “1230199/control_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/control_project/general

[56] M. Ferreira, “1230199/experimental_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/experimental_project/general

[57] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, “ Goal Question Metric (GQM) Approach ,” *Encyclopedia of Software Engineering*, Jan. 2002, doi: 10.1002/0471028959.SOF142.

[58] Apache Software Foundation, “Apache JMeter - User’s Manual: Glossary.” Accessed: Mar. 17, 2025. [Online]. Available: https://jmeter.apache.org/usermanual/glossary.html

[59] P. Virtanen *et al.*, “SciPy 1.0: fundamental algorithms for scientific computing in Python,” *Nat Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, doi: 10.1038/S41592-019-0686-2.

[60] G. Malato, “An Introduction to the Shapiro-Wilk Test for Normality | Built In.” Accessed: Mar. 20, 2025. [Online]. Available: https://builtin.com/data-science/shapiro-wilk-test

[61] H. Motulsky, “GraphPad Prism 10 Statistics Guide - Choosing a normality test.” Accessed: Mar. 20, 2025. [Online]. Available: https://www.graphpad.com/guides/prism/latest/statistics/stat_choosing_a_normality_test.htm

[62] SciPy Team, “normaltest — SciPy v1.15.2 Manual.” Accessed: Mar. 20, 2025. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.normaltest.html

[63] D. Siegle, “t Test | Educational Research Basics by Del Siegle.” Accessed: Mar. 20, 2025. [Online]. Available: https://researchbasics.education.uconn.edu/t-test/

[64] DATAtab Team, “Mann-Whitney U Test: Non-Parametric Hypothesis Testing.” Accessed: Mar. 20, 2025. [Online]. Available: https://datatab.net/tutorial/mann-whitney-u-test

[65] Kepler’s team, “kepler/grafana-dashboards at main · sustainable-computing-io/kepler.” Accessed: Mar. 25, 2025. [Online]. Available: https://github.com/sustainable-computing-io/kepler/tree/main/grafana-dashboards

[66] A. K. Dwivedi, I. Mallawaarachchi, and L. A. Alvarado, “Analysis of small sample size studies using nonparametric bootstrap test with pooled resampling method,” *Stat Med*, vol. 36, no. 14, pp. 2187–2205, Jun. 2017, doi: 10.1002/SIM.7263.

[67] U. C. Bureau, “A Basic Explanation of Confidence Intervals,” Oct. 2021, Accessed: Jun. 18, 2025. [Online]. Available: https://www.census.gov/programs-surveys/saipe/guidance/confidence-intervals.html

Appendix A

Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response description
	Description of the risk	Cause of the risk	Effect on the project	Name of person who monitors the risk	Group sourced rough estimate of how likely this is to occur	Rough estimate of how significant the impact of this risk	Probability multiplied by Impact	What will happen if the risk becomes an issue and no action is taken	Decision made by group on how to respond to this risk (see above in blue)	How do you know it is time to put the response into play
1	Tool Incompatibility	Benchmarking tools fail to work as needed	Delays in testing process	Miguel Ferreira	2	4	8	Testing will halt unexpectedly	Mitigate	Verify tool compatibility during the project setup phase; have backup tools ready for critical processes.
2	Synthetic Data Bias	Limited Realist in synthetic datasets	Results may not represent real-world cases	Miguel Ferreira	3	5	15	Misleading conclusions	Mitigate	Use diverse datasets, including samples from publicly available APIs, to simulate multiple scenarios effectively.
3	Protobuf Learning Curve	Complexity of Protocol Buffers	Incresed development time	Miguel Ferreira	3	4	12	Delays in experimental setup	Mitigate	Allocate time early in the project for team training and use official Protobuf resources/documentation .
4	Network Instability	Unstable Network during testing	Skewed performance results	Miguel Ferreira	2	5	10	Unreliable test results	Mitigate	Use a controlled local environment for testing and benchmarking.
5	Resource Constraints	Limited Access to computing rsources	Restricted testing scope	Miguel Ferreira	3	4	12	Missed opportunities for analysis	Mitigate	Plan experiments efficiently to minimize resource waste, and consider requesting additional computing time or infrastructure if necessary.

Figure 13 - Risk Register Part 1

6	Data Loss	Accidental deletion or corruption of data	Rework and delays	Miguel Ferreira	2	5	10	Loss of critical results	Avoid	Use version control for data and make regular backups throughout the project lifecycle.
7	Software Bugs	Errors in benchmarking scripts	Skewed or invalid test results	Miguel Ferreira	4	3	12	Results will be unreliable	Mitigate	Perform code reviews and test scripts thoroughly before running full experiments.
8	Time Constraints	Insufficient time allocated for experiments	Compromised quality of results	Miguel Ferreira	3	5	15	Critical steps may be rushed	Mitigate	Use a detailed timeline with milestones to monitor progress and ensure adequate time is allocated for experimentation.
9	Changes in Tools Availability	Open-source tools become deprecated	Project delays	Miguel Ferreira	2	3	6	Rework on tools' compatibility	Mitigate	Continuously monitor tool development lifecycles and maintain a list of alternatives for all critical tools.

Figure 14 - Risk Register Part 2

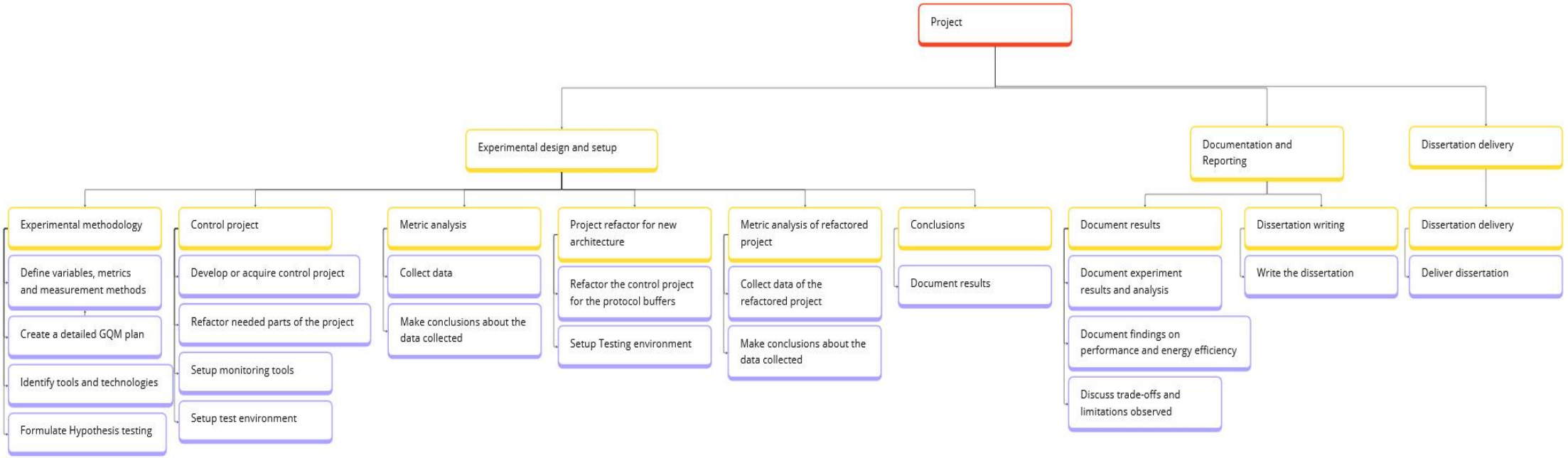

Figure 15 - Work breakdown structure

Table 26 - WBS dictionary

WBS dictionary	Type	Description	Progress criteria
1 Experimental design and setup	Phase	This phase comprises of setting up the projects and acquiring data for analysis.	Experimental methodology – 25% Control project – 25% Metrics analysis – 10% Project refactor – 25% Metrics analysis – 10% Conclusions – 5%
1.1 Experimental methodology	Deliverable	The variables, GQM plan and hypothesis must be defined to have a default execution for each analysis. The advisor must accept the defined variables, GQM plan and hypothesis	Define variables to be analysed – 15% Define measurement methods -20% Create GQM plan – 30% Identify tools and technologies – 15% Define Hypothesis – 20%
1.1.1 Define variables, metrics and measurement methods	Work package	In this work package, the independent and dependent variables should be defined. The metrics to be analysed and the measurement methods should also be defined.	Independent variables – 25% Dependent variables – 25% Define Metrics – 25% Create a standardised Dataset – 5% Review and verify tooling – 20%
1.1.2 Create a detailed GQM plan	Work package	The Goal, questions and metrics should be defined	Goal – 35% Questions – 25% Metrics – 40%
1.1.3 Identify tools and technologies	Work package	Tools and technologies should already be defined, but this work package should review the tools used and anticipate if new tools are needed.	Review tools – 50% Obtain new tools – 50%
1.1.4 Formulate hypothesis testing	Work package		Hypothesis defined – 40% Choose a statistical method – 30% Review and define experiment conditions -30%
1.1.5 Report version 0.1	Work package	Update the dissertation document with the work done in the 1.1 deliverable and review the literature.	Updated document – 100%
1.2 Control project	Deliverable	Acquire and refactor the control project,	Acquire control project – 30% Refactor needed parts – 10% Set up monitoring tools – 30% Set up test environment – 30%
1.2.1 Do a project risk course on Linked	Work package		Finish the course – 100%

1.2.2 Acquire control project	Work package	Acquire a project compatible with the tooling defined previously	Find project – 100%
1.2.3 Refactor needed parts	Work package	Refactor parts of the project that need to be changed	Define refactoring goals – 10% Implement the refactoring -40% Test refactored parts – 50%
1.2.4 Setup monitoring tools	Work package	Install and configure defined monitoring tools	Install monitoring tools – 50% Configure monitoring tools – 50%
1.2.5 Setup test environment	Work package	In this work package, infrastructure and environment configuration should be provided.	Provision infrastructure – 50% Configure environment – 50%
1.2.6 Report version 0.2	Work package	Update the dissertation document with the work done in the 1.1 deliverable and review the literature.	Updated document – 100%
1.3 Metric analysis	Deliverable	This deliverable should have the data collected from the control project and conclusions about that data.	Collect data – 50% Analysis of the data – 25% Conclusion about the data – 25%
1.3.1 Collect data	Work package		Collect data – 100%
1.3.2 Make conclusions about the data collected	Work package		Analysis of the data-30% Conclusion about the data-60% Update document – 10%
1.4 Project refactor for new architecture	Deliverable	Refactor the project to use protocol buffers and change tests to accommodate the protocol buffers.	Refactor project to use protocol buffers – 60% Setup testing – 40%
1.4.1 Refactor the controller project to use protocol buffers	Work package	Refactor the project to use protocol buffers	Define refactoring goals – 10% Refactor to protocol buffers – 85% Test refactored parts – 5%
1.4.2 Setup test environment	Work package	In this work package, infrastructure and environment configuration should be provided, given the latest changes to the project.	Provision infrastructure – 50% Configure environment – 50%
1.5 Metric analysis of refactored project	Deliverable	This deliverable should have the data collected from the refactored project and conclusions about that data.	Collect data – 50% Analysis of the data – 25% Conclusion about the data – 25%
1.5.1 Collect data on refactored project	Work package		Collect data – 100%
1.5.2 Make conclusions about the data	Work package		Analysis of the data-30% Conclusion about the data-60%

			Update document – 10%
1.5.3 Report Version 0.3	Work package	Update the dissertation document with the work done in the two previous deliverables and a review of the literature.	Updated document – 100%
1.6 Conclusion	Deliverable	Make conclusions about the data collected in both projects and document those results.	Make conclusions – 50% Document those results and observations – 50%
1.6.1 Document results	Work package	The document should be updated with the new data collected	Make conclusions – 100%
2 Documentation and reporting	Phase	The observations made previously were further analysed and documented, and an improved version of the dissertation was written.	Analyse the data -30% Document findings – 30% Discuss trade-offs – 20% Improve dissertation – 20%
2.1 Document results	Deliverable	A further analysis was made, and a literature review was also conducted.	Analyse the data – 30% Document findings – 20% Literature review – 50%
2.1.1 Document experiment	Work package		Improve the documentation of the control project data – 50% Improve the documentation of the refactored project data – 50%
2.1.2 Document findings on performance and energy efficiency	Work package	Improve documentation made in response to the GQM plan and hypothesis testing.	Improve analysis of performance against the GQM plan and hypothesis testing – 50% Improve analysis of energy efficiency against the GQM plan and hypothesis testing -50%
2.1.3 Discuss trade-offs and limitations observed	Work package	Document comparison between the two. And ask the advisor for a review	Document comparison – 50% Ask the advisor for approval – 50%
2.2 Dissertation writing	Deliverable	Conclusion of the dissertation, where the document was improved, and a final review of the literature was made	Improve the dissertation – 50% Verify if data is correct – 20% Literature review- 30%
2.2.1 Write the dissertation	Work package		Improve the dissertation 100%
3 Final review and submission	Phase	Review the details and deliver the dissertation	Review and correct the details – 50% Deliver the document – 50%

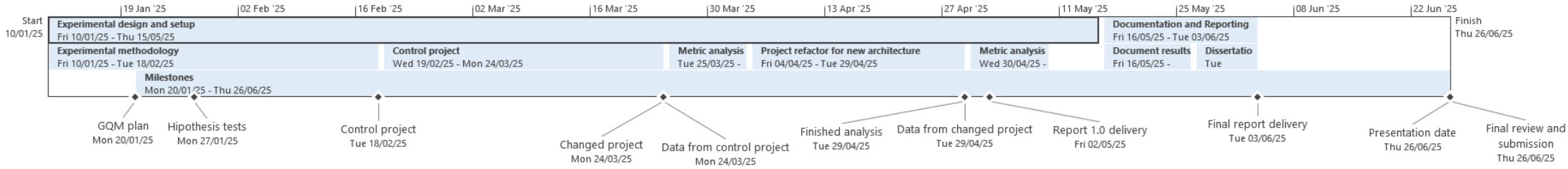

Figure 16 - Full timeline

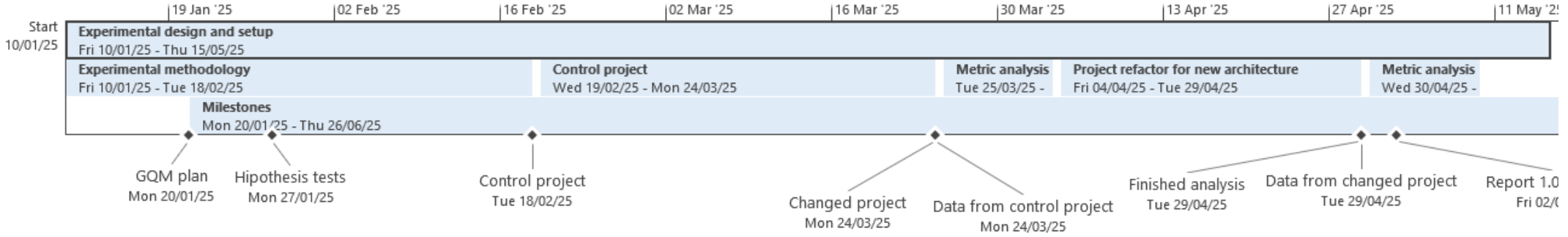

Figure 17 - Improved resolution of timeline part 1/2

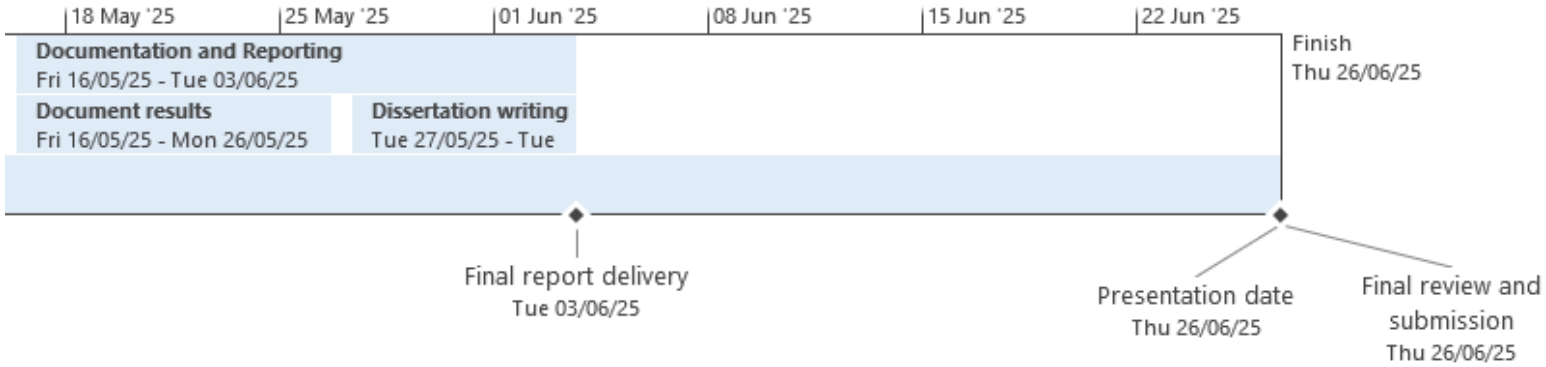

Figure 18 - Improved resolution of timeline part 2/2

Appendix B

Table 27 - Used software for the experiment setup

Software	Version
Minikube	1.35.0
Helm	3.17.1
k6	0.57.0
Node	20.18.0
Npm	10.8.2
Java	17
Maven	3.9.5
Docker	28.0.0
Docker Compose	1.29.2

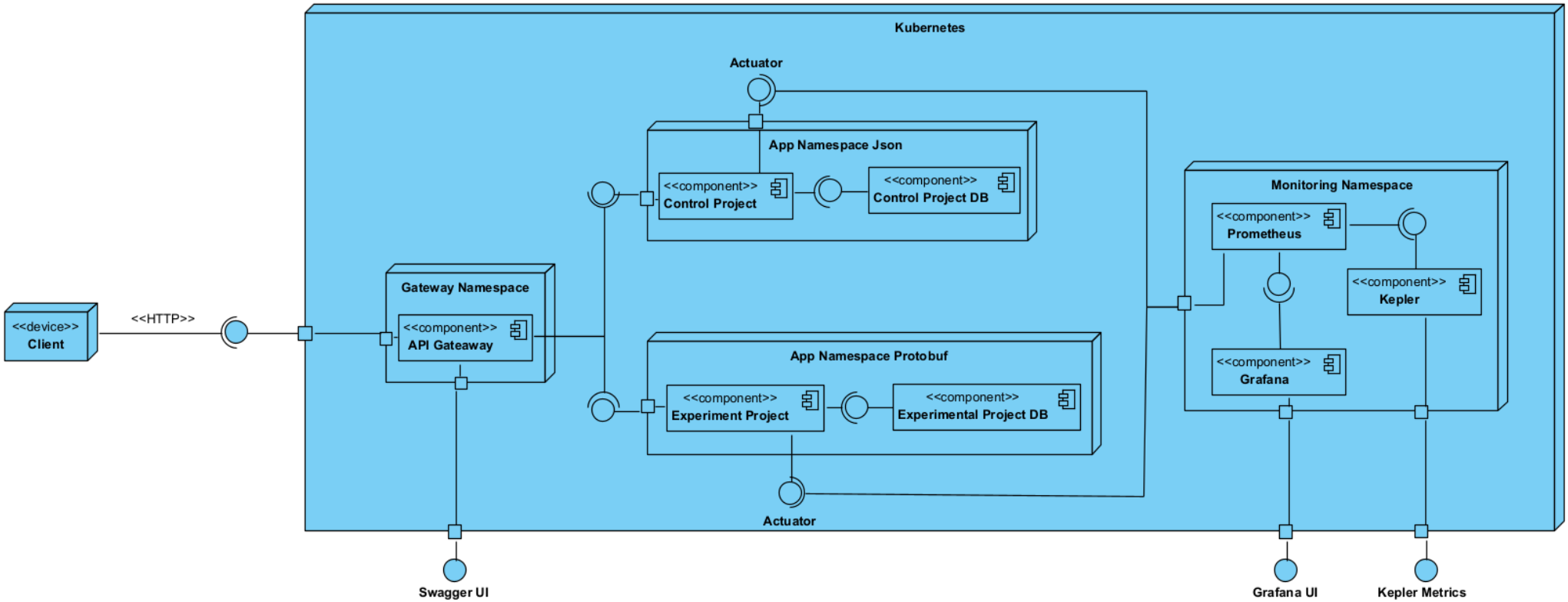

Figure 19 - Deployment diagram

Appendix D

The Code Snippet 19 to Code Snippet 22 show the generated proto files for the user domain.

```
syntax = "proto3";
import "user_roles.proto";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message UserDTO {
  string username = 1;
  string email = 2;
  string password = 3;
  UserRoles role = 4;
}
```

Code Snippet 19 - UserDTO generated proto file

```
syntax = "proto3";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message EditUserDTO {
  string username = 1;
  string email = 2;
}
```

Code Snippet 20 - EditUserDTO generated proto file

```
syntax = "proto3";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message LocalDateTimePb {
  int32 year = 1;
  int32 month = 2;
  int32 day = 3;
}
```

Code Snippet 21 - LocalDateTimePb generated proto file

```
syntax = "proto3";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";


enum UserRoles {
  ADMIN = 0;
  EMPLOYEE = 1;
}
```

Code Snippet 22 - UserRoles generated proto files

As illustrated in Code Snippet 19 to Code Snippet 22, the generation of the proto files for the user domain was facilitated by the IntelliJ plugin. However, the plugin could not automatically generate the packages, nor provide an explicit indication to use the proto3 syntax.

Furthermore, Code Snippet 23 to Code Snippet 25 show the generated protos related to the product entity domain.

```
syntax = "proto3";
import "local_date_time_pb.proto";
import "product.proto";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message ProductEntity {
  int32 id = 1;
  string name = 2;
  LocalDateTimePb created_at = 3;
  LocalDateTimePb updated_at = 4;
  repeated Product products = 5;
}


message ProductEntityCatalog {
  repeated ProductEntity products = 1;
}
```

Code Snippet 23 - ProductEntity generated proto file

```
syntax = "proto3";
import "local_date_time_pb.proto";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message Product {
 int32 id = 1;
 int32 entity_id = 2;
 bool sold = 3;
 LocalDateTimePb created_at = 4;
 LocalDateTimePb updated_at = 5;
}
```

Code Snippet 24 - Product generated proto file

```
syntax = "proto3";
package com.conveniencestore.conveniencestore.Protocol Buffers;
option java_package = "com.conveniencestore.conveniencestore.Protocol
Buffers";

message ProductEntityDTO {
  string name = 1;
}
```

Code Snippet 25 - ProductEntityDTO generated proto file

As with the user domain, the product entity domain generated proto files are visible in Code Snippet 23 to Code Snippet 25. One salient aspect to note is the product proto, which, even though it does not directly belong to the product entity domain, must be created because the product entity has a many-to-one relationship with it.

Appendix E

Table 28 - Average payload size (in bytes) for requests and responses across different endpoints, comparing JSON and Protocol Buffers serialisation

Endpoint / Serialisation	Get All Product Entities (B)		Get All Users (B)		Get User By (B)		Create User (B)		Update User (B)		Delete User (B)	
	Request	**Response**	**Request**	**Response**	**Request**	**Response**	**Request**	**Response**	**Request**	**Response**	**Request**	**Response**
JSON	0	7138	0	63574	0	150	115	167	69	169	0	169
Protocol Buffers	0	1569	0	20849	0	46	48	57	38	61	0	61