

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

**Miguel Alves Ferreira**

**Dissertation**

**Masters in informatics engineering**

**Specialisation in Software Engineering**

Integrity Statement

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

I have not plagiarised or applied any form of misuse of information or falsification of results throughout the process that led to its elaboration.

Therefore, the work presented in this document is original and of my authorship and has not been used previously for any other purpose.

I also declare that I am fully aware of P.PORTO’s Code of Ethical Conduct.

ISEP, Porto, 24 de maio de 2025

Summary

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

Abstract

**Keywords**: JSON, Protocol Buffers, REST, serialization, performance, 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.

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

1 Introduction 1

1.1 Background and Context 1

1.2 Problem Description 1

1.3 Objectives and Research Questions 2

1.4 Research Methodology 3

1.5 Ethical Considerations 3

1.6 Document Structure 4

2 Background 6

2.1 REST 6

2.1.1 How are REST Applications Structured 8

2.2 JSON 9

2.3 Protocol Buffers 11

2.3.1 Structure of Protocol Buffers 11

2.3.2 Schema Compilation and Code Generation 13

2.4 Performance and Energy Analysis Tools 13

2.4.1 Grafana K6 13

2.4.2 Kepler 14

3 Planning 15

3.1 Project Charter 15

3.1.1 Stakeholders 15

3.1.2 Scope, Objectives 16

3.1.3 Benefits 17

3.1.4 Deliverables 17

3.1.5 Time 17

3.1.6 Costs 18

3.1.7 Assumptions, Restrictions and Risks 18

3.2 Work Breakdown Structure 19

3.3 Work Breakdown Structure Dictionary 20

3.4 Timeline 23

3.5 Skills 24

3.5.1 Required Skills 24

3.5.2 Strengths 24

3.5.3 Weaknesses 25

3.5.4 Technical Skills 25

3.5.5 Plans for Improvement 27

4 Literature Review 28

4.1 Research Questions 28

4.2 Data Sources 28

4.3 Search Terms 29

4.4 Eligibility Criteria 30

4.5 Data Collection Process 30

4.6 Discussion 31

4.6.1 RQ1: How does using Protocol Buffers impact the performance of HTTP-based REST architectures? 32

4.6.2 RQ2: To what extent can Protocol Buffers improve energy efficiency in HTTP-based REST architecture communications compared to JSON? 36

4.7 Conclusion 38

5 Experiment 39

5.1 Project selection and analysis 39

5.2 Migration process 40

5.2.1 Components 40

5.2.2 Strategy 41

5.3 Migration 42

5.3.1 Dependencies and Plugins 42

5.3.2 User Domain Migration 45

5.3.3 Product Entity Domain Migration 47

5.3.4 Kubernetes Cluster Setup 48

5.4 Analysis 51

5.4.1 Goal Question Metric 52

5.4.2 Performed Experiments 54

5.4.3 Performance Tests Setup 54

5.4.4 Energy Consumption Tests 65

5.5 Conclusion 71

6 Conclusion 73

6.1 Accomplishments 73

6.2 Difficulties 73

6.3 Threats to Validity 74

List of Figures

[Figure 1 - REST API in action 6](#_Toc198904669)

[Figure 2 - Overall compliance with the REST principles of 500 REST APIS 7](#_Toc198904670)

[Figure 3 - Timeline part 1/2 23](#_Toc198904671)

[Figure 4 - Timeline part 2/2 23](#_Toc198904672)

[Figure 5 - Prisma systematic methodology 30](#_Toc198904673)

[Figure 6 - Average serialization time 32](#_Toc198904674)

[Figure 7 - Average deserialization time 32](#_Toc198904675)

[Figure 8 - Response time benchmark of a GET request uncompressed 33](#_Toc198904676)

[Figure 9 - Benchmark of payload size based on flat data 34](#_Toc198904677)

[Figure 10 - Memory analysis of JSON serialization 34](#_Toc198904678)

[Figure 11 - Memory analysis of Protocol Buffers serialization 35](#_Toc198904679)

[Figure 12 - Energy expended on CPU with data synchronization (volume1) 36](#_Toc198904680)

[Figure 13 - Energy expended on CPU with data synchronization (volume2) 36](#_Toc198904681)

[Figure 14 - Representation of possible GQM approach 51](#_Toc198904682)

[Figure 15 - Work Breakdown Structure 79](#_Toc198904683)

[Figure 16 - Full timeline 80](#_Toc198904684)

[Figure 17 - Improved resolution of timeline part 1/2 80](#_Toc198904685)

[Figure 18 - Improved resolution of timeline part 2/2 81](#_Toc198904686)

[Figure 19 - Risk Register Part 1 82](#_Toc198904687)

[Figure 20 - Risk Register Part 2 83](#_Toc198904688)

List of Tables

Table 1 - List of Types that can be defined in the .proto file 12

Table 2 - Power and interest matrix 15

Table 3 - WBS dictionary 20

Table 4 - Technical Skills needed to complete the dissertation 26

Table 5 - Data sources 29

Table 6 - Average serialization time in ms. Reprinted from [1] 32

Table 7 - Average deserialization time in ms. Reprinted from [1]. 32

Table 8 - Testing hardware specification 51

Table 9 - Question and metrics for performance 52

Table 10 - Question and metrics for energy consumption 53

Table 11 - Endpoints requested in the tests 55

Table 12 - Benchmark results for creating users 56

Table 13 - Benchmark results for retrieving all users 56

Table 14 - Benchmark results for retrieving user by id 58

Table 15 - Benchmark results for retrieving all product entities 59

Table 16 - Benchmark results for updating a user 60

Table 17 - Benchmark results for deleting a user 61

Table 18 - Hypothesis tests for each endpoint in terms of performance 62

Table 19 - Energy consumption of user creation in joules 65

Table 20 - Energy consumption of the retrieval of all users in joules 66

Table 21 - Energy consumption of the retrieval of a user by its id in joules 67

Table 22 - Energy consumption of the retrieval of all product entities in joules 68

Table 23 - Energy consumption of updating a user in joules 69

Table 24 - Energy consumption of deleting a user in joules 69

Table 25 - Hypothesis tests for each endpoint in terms of energy consumption 70

Table 26 - “A list of space-efﬁciency benchmark publications that involve JSON[…]”[2] 87

List of Code Snippets

[Code Snippet 1 - Example of a User JSON 8](#_Toc198904715)

[Code Snippet 2 - Example of Decoding and Encoding a User JSON in Go 9](#_Toc198904716)

[Code Snippet 3 - Block of code from a .proto file defining a User 11](#_Toc198904717)

[Code Snippet 4 - Search query developed with the search terms and research questions 28](#_Toc198904718)

[Code Snippet 5 - Changes to allow Protocol Buffers in project 43](#_Toc198904719)

[Code Snippet 6 - Dependencies to compile Protocol Buffers automatically 43](#_Toc198904720)

[Code Snippet 7 - User response DTO 44](#_Toc198904721)

[Code Snippet 8 - Generated user response Protocol Buffers file 44](#_Toc198904722)

[Code Snippet 9 - Example of one endpoint that uses Protocol Buffers 45](#_Toc198904723)

[Code Snippet 10 - Example of service level insert method using Protocol Buffers 46](#_Toc198904724)

[Code Snippet 11 - Helper methods to convert special data types into Protocol Buffers 47](#_Toc198904725)

[Code Snippet 12 - Example of one endpoint for Product Entity domain 48](#_Toc198904726)

[Code Snippet 13 - Example of service level method to retrieve all Product Entities with Protocol Buffers 48](#_Toc198904727)

[Code Snippet 14- Docker file for both projects 49](#_Toc198904728)

[Code Snippet 15 - Service monitor manifest file for API 49](#_Toc198904729)

[Code Snippet 16 - Deployment manifest file for API 50](#_Toc198904730)

[Code Snippet 17 - Response time hypothesis test structure 62](#_Toc198904731)

[Code Snippet 18 - UserDTO generated proto file 87](#_Toc198904732)

[Code Snippet 19 - EditUserDTO generated proto file 87](#_Toc198904733)

[Code Snippet 20 - LocalDateTimePb generated proto file 87](#_Toc198904734)

[Code Snippet 21 - UserRoles generated proto files 87](#_Toc198904735)

[Code Snippet 22 - ProductEntity generated proto file 88](#_Toc198904736)

[Code Snippet 23 - Product generated proto file 88](#_Toc198904737)

[Code Snippet 24 - ProductEntityDTO generated proto file 88](#_Toc198904738)

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*

**FTP** *File Transfer Protocol*

**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*

**JSON** *JavaScript Object Notation*

**JVM** *Java Virtual Machine*

**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*

# Introduction

The main purpose of this chapter is to provide the background and context and to describe the problem. It also presents the research objectives, and the ethical considerations involved. Finally, it presents the structure of the report.

## Background and Context

REST (Representational State Transfer) APIs (Application Programming Interfaces) 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 [1], [2].

In contrast, Protocol Buffers is a binary serialisation format developed by Google that offers 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 [3], [4], [5]. Despite these advantages, Protocol Buffers' complexity and lack of human readability might limit its adoption in REST architectures, where JSON remains the dominant choice [6].

## Problem Description

Choosing the right technology for data serialisation plays a key role in the performance and scalability of REST applications, as it will be integrated from start to finish of the project [7]. 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 HTTP REST application. Normally, Protocol Buffers are used with Google Remote Procedure Call (gRPC). Which typically uses Protocol Buffers over HTTP/2, is often observed to have performance advantages over traditional RESTful APIs implemented over HTTP/1.1 [8], [9].

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 [10], which is about making proper design decisions in the life cycle of the system so that it has a reduced carbon footprint, as such, increased latency, reduced throughput, or inefficient resource usage can significantly impact user experience and system scalability, on top of that, research has shown that underutilized or poorly optimized systems lead to higher costs, resource wastage, and scalability issues [11], [12]. For this reason, efficient 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 [12], [13], for example, in modern data centres, serialisation optimisations can lead to substantial energy savings, translating into lower costs and reduced environmental impact [12].

## Objectives and Research Questions

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

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

## Research Methodology

The research methodology chosen for this study is based on controlled experiment research. 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 [14], [15]. 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 [16].

To ensure accuracy and credibility, the research incorporates measurement mechanisms such as repeated trials and 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.

## Ethical Considerations

This study will adhere to ethical principles to ensure integrity and transparency. Although the research will not involve human subjects or sensitive data related to any subjects, the study will make sure all test data used in the study is synthetic and generated specifically for benchmarking purposes. With no real user data or proprietary information to be used, ensuring compliance with data protection regulations such as the General Data Protection Regulation (GDPR).

Furthermore, the experimental setup, tools, and methodologies have been documented to ensure the replicability of the results by other researchers. To ensure the integrity of performance evaluations, the project is completely devoid of any form of bias. To ensure fair comparisons, both serialization formats should be evaluated under equal conditions whenever possible.

All source code developed for this study is publicly available from the study's inception to its conclusion, in accordance with 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 improving the study.

In conclusion, the code of ethics established by the Institute of Electrical and Electronics Engineers (IEEE) [17] 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 [18].

## Document Structure

The dissertation is comprised of a total of ten sections. The initial section 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, and the ethical considerations that have been taken into account in the development and procedures.

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

The third section, the Planning, 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 section, 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.

Moreover, the Literature Review section 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 subsequent section is entitled Research Methodology, which provides insight into the research methodology that was employed, as well as the way the data collection was to be carried out.

The subsequent three chapters, namely, the Analysis and Migration, Implementation and Experiment, address the project selection for the dissertation, the alterations applied to the base project selected, the implementation details of the migration, and the experiment performed on the project with data collection and a discussion on the data collected.

To summarise, 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.

# Background

The subsequent section aims to provide information on the fundamental concepts related to the work that will be developed. The concepts that will receive the most attention are the REST architectural style and the two serialisation formats mentioned above, in this case, JSON and Protocol Buffers.

## REST

REST is an architectural style introduced in 2000 [19]. It defines a set of standards and constraints for designing distributed systems, especially web services. It also provides a lightweight stateless client-server communication model that can enable an application to be scalable, efficient, and easy to maintain, these are contributing factors to its wide usage, not only that, its simplicity and compatibility with web protocols make it adaptable to a wide variety of clients, from web browsers and mobile phones to complex IoT systems. As a result, REST principles are foundational in contemporary architecture, such as microservices and cloud computing [20].

A diagram of a rest api

Description automatically generated

Figure 1 - REST API in action

Reprinted from [20]

At its core, REST leverages the Hypertext Transfer Protocol (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.

Data exchanged in RESTful systems can be serialised in various formats, such as JSON, XML, or Protocol Buffers. This flexible design allows developers to choose the most suitable format for their specific use case. Arguably, REST offers significant benefits such as scalability, flexibility, and minimal maintenance costs, but it has some limitations, like:

* High Latency: Multiple client-server interactions may increase response times.
* High Bandwidth Usage: Particularly when verbose formats like JSON or XML are used.
* High Energy Consumption: Significant processing power may be required for encoding, decoding, and data transfer.

### How are REST Applications Structured

Restful 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 [21], [22]:

* 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.

A graph with numbers and a bar

Description automatically generated

Figure 2 - Overall compliance with the REST principles of 500 REST APIS

Reprinted from [6]

Although REST has become the dominant architectural style for web services, studies show that many implementations deviate significantly from its core principles, as we can see in Figure 2. 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 [6].

## 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 [23].

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.

Below is an example of a JSON object that demonstrates these 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 [1], [3], [5]. 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 JSON, 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. The following example shows how 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 this Go example, 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 string.

## Protocol Buffers

Protocol Buffers, commonly referred to as Protocol Buffers, 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" [24]. Protocol Buffers is 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 [5], [25], [26], [27].

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[8], [28]. 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 RESTful architectures [4], [28], [29]. 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.

### 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. The following example 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 user

In the Schema:

* 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-points |

The table depicted above shows the types that Protocol Buffers supports.

### 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 for interacting with the data, these files typically have a .pb.go extension in Go projects.

## Performance and Energy Analysis Tools

Using performance and energy analysis tools is imperative for assessing the efficiency and resource utilisation of software systems. These tools allow us to observe applications under diverse workloads and operational conditions. In this dissertation, two analytical tools are employed: Grafana k6 and Kubernetes Efficient Power Level Exporter (Kepler).

### Grafana K6

Grafana k6 is a versatile, open-source performance testing tool developed to evaluate the performance and reliability of web applications and services. It can simulate concurrent user requests to a target server, enabling developers and testers to analyse system behaviour under variable load conditions. It has support for multiple protocols, including HTTP, Hypertext Transfer Protocol Secure (HTTPS), and File Transfer Protocol (FTP), which enhances its utility as a comprehensive performance-testing solution.

**Main Types of Tests:**

* **Load Test**: “Load testing focuses on the ability of a system to handle increasing levels of anticipated realistic loads resulting from transaction requests generated by controlled numbers of concurrent users or processes.” [30]
* **Stress Test**: “Stress testing focuses on the ability of a system or component to handle peak loads that are at or beyond the limits of its anticipated or specified workloads. Stress testing is also used to evaluate a system’s ability to handle reduced availability of resources such as accessible computing capacity, available bandwidth, and memory.” [30]
* **Endurance Test**: “Endurance testing focuses on the stability of the system over a time frame specific to the system’s operational context. This type of testing verifies that there are no resource capacity problems (e.g., memory leaks, database connections, thread pools) that may eventually degrade performance and/or cause failures at breaking points.” [30]
* **Spike Test**: “Spike testing focuses on the ability of a system to respond correctly to sudden bursts of peak loads and return afterwards to a steady state.” [30]

### Kepler

Kepler 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 [31], [32]:

* **Data collection via EBPF and Hardware Counters:** This allows Kepler to directly collect performance metrics from the Linux kernel, and due to utilising hardware counters, it is also capable of gathering detailed information on energy consumption estimates [31], [32].
* **Real-time power consumption metrics:** It can access data from hardware components through their API to get accurate and real-time power metrics [31], [32].
* **Power consumption attribution:** It applies to the ratio power model to attribute power usage to individual processes. This model calculates the proportion of a process’s resource utilisation relative to the entire system. As such, by mapping process IDs to container and pod identifiers, Kepler can aggregate power consumption metrics, providing insight at various levels of the Kubernetes hierarchy.

# Planning

The current chapter describes the project planning, which will be crucial to finishing the project on time and having palpable dates to define what has already been done and what needs to be done.

## Project Charter

The project charter is a living document that should represent the brief planning of the project.

### Stakeholders

This project provides valuable insights for diverse stakeholders, who may benefit in several ways from the knowledge presented. Table 2 outlines the defined stakeholders.

Table 2 - Power and interest matrix

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

Stakeholder Roles:

* Developers: Developers benefit from the project’s insights into serialisation technologies, enabling them to make better technical decisions.
* Software Development Companies: These companies aim to reduce costs and improve performance. The findings may influence their decisions about adopting Protocol Buffers within REST architecture.
* Students: The project will serve as a valuable learning resource, helping students learn more about serialization and the impact it has on a software system.
* Researchers: The findings may inspire further academic studies.
* Advisor: The advisor will ensure the project aligns with academic standards, providing guidance and assistance in overcoming challenges.

### 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. On the other hand, Protocol Buffers is a schema-driven binary serialisation format, which offers significant performance advantages like reduced payload sizes, improved parsing speeds, and optimised resource consumption. Some of the problems with Protocol Buffers are often connected to their readability and complexity.

With the increased demand for scalable systems, information like energy efficiency and high performance is crucial for microservices and IoT applications, as such, obtaining insights on how serialisation formats might influence these aspects in a system is important. Furthermore, the research explores the trade-offs between JSON and Protocol Buffers within REST architectures, additionally, it analyses key metrics, such as latency, resource usage, and energy efficiency, giving the necessary insights to optimise communication in software systems.

The primary objective of this thesis is to perform a comprehensive evaluation of Protocol Buffers in REST architectures and compare key metrics against JSON, these metrics are:

* **Performance**: Analyse the response time and throughput for both formats.
* **Scalability**: See how well each format performs under different loads and data sizes.
* **Energy Consumption**: View the energy utilisation during serialisation and deserialization.

These insights are intended to assist developers and architects with actionable advice on whether to use Protocol Buffers in RESTful apps.

### Benefits

The benefits of the current research are:

* **Cost Reduction:** If Protocol Buffers demonstrates lower energy consumption or faster performance, this could translate into reduced bandwidth and operational costs.
* **Improved Application Performance:** Faster serialisation and smaller payload sizes might improve response times, which is critical for performance-sensitive applications.
* **Advancements in Software Development:** The research could encourage broader adoption of Protocol Buffers and inspire innovation in serialisation methods.
* **Scalability Insights:** Findings will help companies and developers better scale their systems by understanding serialisation impacts.

### Deliverables

The project aims to produce the following deliverables:

* **Project Plan:** Detailed timeline and task breakdown.
* **Dissertation Report:** A thesis that documents the study, techniques, findings, and interpretations.
* **Software Code:** All Code related to the experimental setup, including benchmarks and test scripts.
* **Testing Reports:** Detailed analysis of performance metrics such as latency, throughput, and energy consumption.
* **Data Collected:** Processed and raw data from benchmark experiments.
* **Presentation and Discussion:** A final presentation summarising the project’s key findings, to be delivered to stakeholders.

### Time

These are the following milestones and mandatory dates for the project. The milestones can and will change due to a lack of knowledge of the processes, but the mandatory dates will hopefully stay the same throughout 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 – 30/04/2025
* Presentation Date – 26/06/2025

**Mandatory Dates:**

* Prepd review delivery – 06/12/2024
* Report and Presentation Delivery – 04/01/2024
* Final Delivery – 26/06/2024

### Costs

No direct costs have been 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 are expected. These are managed by scheduling ample time for setup and practice.

### Assumptions, Restrictions and Risks

**Assumptions:**

* Benchmarking tools and required libraries will be accessible.
* Experimental environments will be consistent across tests.
* Digital libraries for literature review will remain accessible.

**Restrictions**

* Access to real-world datasets is limited, the study relies on synthetic data for experiments.
* Resource limitations may restrict the scale of experiments (e.g. server capacity or cloud computing costs).
* Benchmarking tools may impose constraints on the range of metrics that can be measured (e.g. some tools may not support energy profiling).
* Tests are conducted in controlled environments, not reflecting a real-world scenario.
* A fixed timeline may limit the number of experiments done.

Risk management is going to be done to ensure that potential threats are identified, assessed, and mitigated, this should ensure that there won’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 19 and Figure 20 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 [33]. Below are some examples:

* **High-Priority Risks**:
  + Synthetic data bias (Probability and Impact Score: 15)
  + Time constraints limiting experimental scope (Probability and Impact Score: 15)
* **Medium-Priority Risks**:
  + Tool incompatibility (Probability and Impact Score: 8)
  + Data loss due to accidental corruption (Probability and Impact Score: 10)

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

## Work Breakdown Structure

The work Breakdown structure is in Appendix A, Figure 16, 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, and data from the experiments are analysed to identify trends, draw conclusions, 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.

## Work Breakdown Structure Dictionary

This section provides a comprehensive examination of the detailed work breakdown structure (WBS) dictionary. The following table 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.

Table 3 - WBS dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| **WBS dictionary** | **Type** | **Description** | **Progress criteria** |
| 1 Experimental design and setup | Phase | This phase will comprise 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 Do the two LinkedIn leadership courses | Work package |  | Finish the course – 100% |
| 1.1.2 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% |
| 1.1.3 Create a detailed GQM plan | Work package | The Goal, questions and metrics should be defined | Goal – 35%  Questions – 25%  Metrics – 40% |
| 1.1.4 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% |
| 1.1.5 Formulate hypothesis testing | Work package |  | Hypothesis defined – 40%  Choose a statistical method – 30%  Review and define experiment conditions -30% |
| 1.1.6 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 are going to be further analysed and documented, and an improved version of the dissertation is going to be written. | Analyse the data -30%  Document findings – 30%  Discuss trade-offs – 20%  Improve dissertation – 20% |
| 2.1 Document results | Deliverable | A further analysis is going to be made, a literature review is also going to be made. | 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 will be improved, and a final review of the literature is going to be 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% |

## Timeline

The full timeline is in Appendix A Figure 16. Below, in Figure 3, we can see the planned timeline, for a better resolution, seek Figure 17 in Appendix A, the same applies to the second image below, seek Figure 18 to see a better image resolution.

A close-up of a computer screen

Description automatically generated

Figure 3 - Timeline part 1/2

A screenshot of a computer

Description automatically generated

Figure 4 - Timeline part 2/2

## 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.

### 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 explicated are important to ensure the project's success.

### Strengths

By doing some self-reflection, I have identified some aspects in which I am good. These strengths will favour the project and my personal development:

* **Lifelong Learning**: I have a mindset that views life as a continuous journey of discovery and growth, enabling me to overcome challenges and remain curious and open to acquiring knowledge.
* **Adaptability and Flexibility**: I am quick to adjust to new circumstances without much trouble.
* **Stress Management**: I perform well under pressure, channelling stress into productivity and maintaining focus during challenging periods.

### Weaknesses

Conversely, I have identified several skills where improvement is needed. These weaker areas have posed challenges in the past and could hinder my effectiveness if not addressed:

* **Leadership**: I am uncomfortable assuming leadership roles. My hesitation stems not from stress but from the demands of multitasking and acting as a central point of communication and coordination.
* **Decision making**: Closely tied to my difficulties with leadership, decision-making feels daunting due to my fear of potential risks and the consequences of incorrect choices.
* **Writing Communication**: I struggle with crafting well-structured and coherent written texts. My tendency to focus solely on the message, without considering clarity or readability, often makes my writing difficult to understand.

### Technical Skills

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

Table 4 - 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. |
| Prometheus | 7 | 3 | With mild knowledge of the tool, it was only used for Kubernetes to gather simple usage data. For this project, its use is going to be central to extracting data, and it needs further review. |
| Grafana | 7 | 1 | No prior knowledge of Grafana, and as such, requires a lot of learning. |
| 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 will require 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 4 shows the needed skills to be able to successfully 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.

### Plans for Improvement

To address these weaknesses, a detailed improvement plan has been devised, tailored to each area:

* **Leadership and Decision-Making**: To develop my leadership skills, I have enrolled in two LinkedIn courses on leadership principles [34], [35]. Additionally, to enhance my decision-making abilities I have enrolled in a risk-specific course [36]. These courses provide insights into project management, risk assessment, and effective decision-making processes. By understanding how to evaluate risks and implement strategies to mitigate them, I aim to reduce my fear of making decisions and build up my confidence.
* **Written Communication:** To improve my writing, tools such as Grammarly are used to refine my grammar and expand my vocabulary. Furthermore, I plan to review scientific articles in areas that I am enthusiastic about, this allows me to improve my writing skills and strengthen my critical thinking.
* **Technical Skills:** Improving technical skills in our field typically involves two key approaches: studying documentation and firsthand application of the technology. However, given that direct application is not currently feasible, the primary focus will be on thoroughly studying relevant documentation. This approach will serve to build proficiency with tools I am less familiar with while also allowing me to close the knowledge gap for tools I already have some experience with but need to master further.

# 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 will outline 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.

## 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?

## 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 5 - 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.

## Search Terms

The keywords identified for the problem described are as follows:

* Protocol Buffers
* Protobufs
* JSON
* REST
* Performance
* Serialization
* Efficiency
* Resource consumption
* Energy consumption

With these keywords, a search query was created:

("Protocol Buffers" **OR** “Protobufs”) **AND** "REST"

**AND** ("serialization" OR "deserialization")

**AND** ("JSON" **OR** "XML")

**AND** ("performance" **OR** "latency" **OR** "resource consumption" **OR** "efficiency" **OR** “energy consumption”)

**AND** ("web applications" OR "mobile" **OR** "IoT" **OR** "microservices")

**AND** ("experimental study" **OR** "benchmark" **OR** "simulation")

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 deserialization 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 metrics like latency, scalability, efficiency, resource consumption, and energy consumption were included to focus on studies evaluating system optimisation. To ensure the inclusion of relevant empirical evidence, terms such as "experimental study," "benchmark," and "simulation" were added.

## 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.

## Data Collection Process

The Prisma systematic methodology [37]is being used 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 are going to be evaluated then after an analysis and if they are relevant to the research topic and research questions they will be included in the review.
* Inclusion: All the studies that are relevant to the research questions and objectives will be included in the review.

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 5 shows the Prisma flowchart.

A flowchart of records

Description automatically generated

Figure 5 - Prisma systematic methodology

Adapted from [38]

## 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, we may witness a shift in serialisation standards with binary serialisation being embraced.

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

Protocol Buffers are widely recognised for their performance and efficiency, offering advantages that make them particularly well-suited for high-performance applications. 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 [3], [4], [5].

A comprehensive evaluation by Juan Cruz Viotti and Mital Kinderkhedia [2] demonstrated that schema-driven serialisation formats, such as Protocol Buffers, consistently outperform schema-less formats like JSON in terms of space efficiency. Even when JSON was compressed, Protocol Buffers, also compressed, maintained its superiority in reducing data size, highlighting its adaptability to constrained environments. This phenomenon can be observed in Table 26 which can be found in Appendix B, which shows the different studies comparing different serialization formats and what these studies concluded.

Audie Sumaray and S. Kami Makki[1]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 6 - Average serialization time in ms. Reprinted from [1]

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

Table 7 - Average deserialization time in ms. Reprinted from [1].

|  | **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 |

A graph of different types of data

Description automatically generated

Figure 6 - Average serialization time

Reprinted from [1]

A graph of different colored bars

Description automatically generated with medium confidence

Figure 7 - Average deserialization time

Reprinted from [1]

From Table 6 and Table 7, a significant discrepancy in serialization and deserialization speeds across the different formats can be observed. XML consistently demonstrates the highest average times for both serialization and deserialization, indicating its relative inefficiency, with that, we can see that 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.

Figures 6 and 7 further corroborate these findings, in Figure 6, the serialization 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 serialization formats. Moreover, JSON is a middle ground between the other serialization 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[39] 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"[26]. This research evaluates Protocol Buffers within the context of RESTful microservices, specifically targeting Quarkus-based, cloud-native architectures. Not only that but also highlights Protocol Buffers advantages in serialization efficiency and reliability, especially when computer resources are scarce. Additionally, it demonstrated a significant performance improvement in serialization 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% smaller in the best-case scenario, this revelation further emphasizes Protocol Buffers' capacity to optimize 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 serialization often failed to complete within allocated memory limits for highly nested or large data structures, whereas Protocol Buffers successfully serialized data, leveraging its efficient binary encoding.

A graph of a number of individuals

Description automatically generated with medium confidence

Figure 8 - Response time benchmark of a GET request uncompressed

Reprinted from [26]

From Figure 8, it is evident that 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 serialization formats. This observation suggests that while Protocol Buffers are inherently more efficient, caching can serve as an equalizer under certain conditions, reducing the impact of serialization inefficiencies on overall response time.

A graph of data generation

Description automatically generated

Figure 9 - Benchmark of payload size based on flat data

Reprinted from [26]

Figure 9 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.

A graph showing the amount of a number of objects

Description automatically generated with medium confidence

Figure 10 - Memory analysis of JSON serialization

Reprinted from [26]

A graph showing the number of the same size

Description automatically generated with medium confidence

Figure 11 - Memory analysis of Protocol Buffers serialization

Reprinted from [26]

Figures 10 and 11 provide deeper insights into the memory profiles of the two formats during serialization. Figure 11 reveals that Protocol Buffers consistently complete the serialization process with significantly lower memory usage, showcasing their streamlined binary encoding mechanism when dealing with increased data count. In contrast, Figure 10 demonstrates that JSON serialization 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.

### 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,"[3] provides an insightful analysis of the energy and performance implications of using three different data interchange formats such as Protocol Buffers, JSON, and XML on mobile devices. The research primarily focuses on mobile applications that frequently synchronize data with web servers, such as backup systems and monitoring tools, where energy consumption is a critical factor due to battery constraints, and as such some metrics, like energy consumption, synchronization speed, and the impact of compression, using two network interfaces (Wi-Fi and 3G) were analyzed. Their findings revealed that Protocol Buffers generally surpass in terms of synchronization time and energy efficiency, especially in large data volumes. However, when compression is applied to text-based formats like JSON and XML, it significantly narrows the performance gap. Compression reduced the size of text-based payloads by approximately 66%, enhancing their performance on slower network interfaces like 3G, where the overhead of data transmission becomes more apparent.

Protocol Buffers showed distinct advantages in scenarios involving uncompressed data or raw binary payloads, such as multimedia transfers, where their ability to encode binary data directly is unmatched by text-based formats. The binary nature of Protocol Buffers allows for smaller data sizes and faster processing times during serialization and deserialization, which may allow for reduced CPU workload on mobile devices. Hence, when applications might require speed and raw efficiency, Protocol Buffers appears to be the superior choice. However, this efficiency comes with trade-offs, it was found that Protocol Buffers, while faster in synchronization and processing, required more CPU energy for their operations compared to compressed JSON. This raises a critical consideration for mobile developers: whether to prioritize faster data processing or minimize energy consumption, particularly in scenarios where battery life is paramount.

A graph of different colored bars

Description automatically generated with medium confidence

Figure 12 - Energy expended on CPU with data synchronization (volume1)

Reprinted from [3]

A graph of a bar chart

Description automatically generated with medium confidence

Figure 13 - Energy expended on CPU with data synchronization (volume2)

Reprinted from [3]

Both Figures 12 and 13 show that Protocol buffers do indeed expend more energy on the CPU.

## 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 serialization 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 its applicability, in this case, its 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 prioritize performance and scalability, nonetheless, JSON is still the preferred choice when simplicity, flexibility is 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.

# 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.

## 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[40], 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 a 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 Springboot project under consideration enables the management of a convenience store, incorporating a range of products, users, and transactions involving products purchased by users. 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 is also noteworthy that the project incorporates 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 be impartial about the project, the changes made needed to be as minimal as possible, as such, only two key changes were made:

* The first one was the database, which was already an in-memory database, in this case, HSQLDB, lacked a way to debug it, because of that, a change was made to have the H2 in-memory database, which is easier to debug.
* Lastly, the Prometheus and the actuator dependencies were added to enable Prometheus to scrape metrics from the HTTP REST API

## Migration process

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

It is important to note, once again, that, in order 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 in the event that 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 on the analysis being conducted.

### Components

The process of selecting a component essentially consisted of interpreting the topic of the dissertation, like selecting components for which the serialization 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 to this analysis. However, components that exhibited less intensive serialization were also selected to ascertain whether, in all situations, one serialization 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 most fields requiring a lot of serialization. Consequently, the serialization method assumes greater significance within the process. Additionally, it encompasses a component of particular interest, namely authentication, which may not be entirely correct to include in this study. However, it is interesting to analyze the impact of serialization 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 serialization 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 serialization format in comparison to another.

### 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 will be elucidated in the subsequent section.

## Migration

The present chapter 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.

### 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 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 will then compile 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 - 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.

### 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 7.1, where the utilisation of an IntelliJ plugin is demonstrated. Code Snippet 8 provides a visual representation of the structure of a proto file. These files are in the src/main/proto directory. The Appendix D illustrates Code snippets 19 to 22, which depict the generated proto files associated with the employed User DTOs.

In order to compile these proto files to Java, it is necessary to execute the compilation lifecycle action in Maven. The files in question will be 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 clearly identified, in this case, we need to specify what it produces and what consumes, in this case, we specify the following “application/x-Protocol Buffers”. Furthermore, to transmit a request body with a Protocol Buffers, it is necessary to specify the data that will be serialized. 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 are not expected to occur in the benchmark that will be made, nor is 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*())).*build*();

}

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

Lastly, in the User domain, introduce the new Protocol Buffers classes into the service level, it's also straightforward, the Protocol Buffers is 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, and as we can see, there is a considerable amount of verbosity. This is particularly evident in the serialization and deserialization 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, then it was compiled into a Java class. Next, changes were made at the controller level, and finally, at the service level.

### Product Entity Domain Migration

The procedure delineated in Section 7.2 is replicated. Initially, proto files are created. Appendix D contains the generated proto files for this domain, as illustrated in Code Snippets 18 to 24. 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.newBuilder().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 Product Entity domain

The same principle is applied here, as in the User domain. As illustrated in Code Snippet 12, we have the changes 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 specialized proto is required for the list. As illustrated in Code Snippet 22, 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, using the same principle as in Section 7.2.

### Kubernetes Cluster Setup

In order to facilitate the seamless execution of tests and the utilization 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 [41], [42]. 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

Subsequent to 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 [43], 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 [44]. 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.

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

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

## Analysis

This section is going to provide insight into the experiments developed and the methodology used to evaluate the comparison between JSON and Protocol Buffers. Furthermore, the GQM approach [45] 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 minimize confounding variables.

Firstly, a controlled environment was established for the execution of the tests. This is important because it minimizes 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, Table 10 presents the hardware specifications of the utilized 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 |

### Goal Question Metric

The methodology 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 in order to answer the questions [45] Figure 14 is a visual representation of how the GQM can look.

A diagram of a question

AI-generated content may be incorrect.

Figure 14 - Representation of possible GQM approach

Reprinted from [45]

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”.

#### Performance

Table 9 - Question and metrics for performance

| **Question** | **Metrics** |
| --- | --- |
| What is the difference in the performance of Protocol Buffers against JSON in HTTP-based REST architectures? | Throughput |
| Response time |

To further evolve the GQM, it is essential to understand the metrics being evaluated, which later will subsequently be compared between the two serialization formats. According to Table 8, two metrics have been defined for measurement, in this case, throughput and response times. Throughput is the number of requests processed within a timeframe [46], 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 serialization might have on the request.

#### Energy Consumption

Table 10 - Question and metrics for energy consumption

| **Question** | **Metrics** |
| --- | --- |
| What is the difference in energy consumption of Protocol Buffers against JSON in HTTP-based REST architectures? | Energy consumption (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 Section 1. According to Table 9, 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 will facilitate a more nuanced understanding of the weight of the serialization method in the requests.

### Performed Experiments

This section presents the results obtained from the experiments that were conducted, the environment in which they were conducted, 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, 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 iterations using the gateway.

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

### Performance Tests Setup

The execution of these tests was done using k6, a tool that uses JavaScript to perform different types of performance tests, the possible tests have already been specified in Section 2.4.1, furthermore, the tool also allows the generation of personalized metrics and is extensible with other packages using Webpack, which was needed for two of the requests [47]. Moreover, the tests performed were load tests, and it was defined that only a single virtual user (VU) would be used and that for each endpoint, it would be sent 10,000 requests. 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 at the script’s directory.

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 domains of user creation, user retrieval, and product entity 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 information update, and user deletion.

The following tables present the analysed data, it contains the executed test case, the number of iterations 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 95.40 | 10.44 | 92.34 | 283.64 | 58.39 |
| Protocol Buffers | 1,000 | 92.26 | 10.76 | 89.58 | 293.71 | 60.75 |
| JSON | 10,000 | 92.32 | 10.79 | 91.60 | 1000.78 | 58.2 |
| Protocol Buffers | 10,000 | 91.57 | 10.84 | 90.86 | 294.35 | 59.1 |
| JSON + Gateway | 1,000 | 98.22 | 10.14 | 93.45 | 275.99 | 59.81 |
| Protocol Buffers + Gateway | 1,000 | 93.22 | 10.66 | 90.23 | 313.70 | 61.21 |
| JSON + Gateway | 10,000 | 93.48 | 10.66 | 92.12 | 286.70 | 59.11 |
| Protocol Buffers + Gateway | 10,000 | 91.36 | 10.87 | 90.95 | 292.11 | 59.20 |

Table 12 shows that Protocol buffers were superior in every analysis performed for the creation of a user with a mere 1,000 iterations, and the absence of a gateway, Protocol Buffers exhibited a 3.30% lower average response time. Additionally, they demonstrated a 3.06% 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

Furthermore, the introduction of a gateway added approximately 3 milliseconds to the average response time at 1,000 iterations, while Protocol Buffers increased only 1 millisecond. It is noteworthy that, after the introduction of the gateway, the performance disparity remained consistent with the prior observation, indicating that with an increase in the number of iterations, the discrepancy between Protocol Buffers and JSON, in this endpoint, diminished.

Table 13 - Benchmark results for retrieving all users

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 5.71 | 165.37 | 2.82 | 81.40 | 1.86 |
| Protocol Buffers | 1,000 | 5.13 | 181.94 | 2.53 | 69.03 | 1.44 |
| JSON | 10,000 | 2.62 | 341.49 | 2.17 | 68.11 | 1.57 |
| Protocol Buffers | 10,000 | 2.27 | 383.61 | 1.98 | 44.89 | 1.05 |
| JSON + Gateway | 1,000 | 6.13 | 153.98 | 4.01 | 70.99 | 2.68 |
| Protocol Buffers + Gateway | 1,000 | 5.80 | 162.02 | 3.61 | 67.91 | 2.31 |
| JSON + Gateway | 10,000 | 3.82 | 241.05 | 3.51 | 62.17 | 2.49 |
| Protocol Buffers + Gateway | 10,000 | 3.49 | 258.05 | 3.32 | 55.44 | 1.91 |

Looking closely at Table 13 it shows that Protocol Buffers have superior performance in comparison to JSON under the same conditions. One explanation for this is due to their increased data requirement for serialization, where Protocol Buffers gains a bigger margin of performance compared to the previously analysed data from Table 12. When performing 1,000 iterations and without a gateway, Protocol Buffers has a 5.71% reduction in average response time, they also have a 10.01% increase in throughput and a 10.28% reduction in median response time, showing a notable difference between the two serializations.

Furthermore, in the absence of a gateway during the 10,000 iterations, the discrepancy persists, resulting in Protocol Buffers exhibiting 13.36% reduction of average response time, 12.33% improved throughput, and 8.76% reduction in median response time.

In conclusion, it was found that the number of iterations performed, whilst using a gateway, resulted in a consistent increase in the elapsed time for both serialisation formats. However, it was observed that Protocol Buffers still showed superior performance in comparison to their JSON counterpart.

Table 14 - Benchmark results for retrieving user by id

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 4.39 | 216.33 | 1.85 | 73.74 | 1.14 |
| Protocol Buffers | 1,000 | 4.86 | 195.64 | 1.97 | 68.94 | 1.08 |
| JSON | 10,000 | 1.71 | 518.92 | 1.45 | 62.64 | 0.84 |
| Protocol Buffers | 10,000 | 1.83 | 487.93 | 1.44 | 54.04 | 0.71 |
| JSON + Gateway | 1,000 | 5.36 | 177.88 | 3.25 | 67.13 | 1.87 |
| Protocol Buffers + Gateway | 1,000 | 6.31 | 151.99 | 3.32 | 65.19 | 2.19 |
| JSON + Gateway | 10,000 | 3.13 | 295.46 | 2.87 | 58.90 | 1.70 |
| Protocol Buffers + Gateway | 10,000 | 2.95 | 312.52 | 2.71 | 54.85 | 1.46 |

The Table 14 results are contrary to those observed in the other two tables. When performing 1,000 iterations without a gateway, a substantial discrepancy is observed, showing that Protocol Buffers have a 10.71% increase in the average response time, the throughput follows the same results, with a 9.56% reduction in the requests per second and finally, the median response time also has a 6.49% increase, this performance maintains itself in the 10,000 iteration test case without a gateway, but it already shows slight signs of Protocol Buffers improving over time, even to the point of the median being just slightly better than the JSON one.

Furthermore, the difference when a gateway is present the differences are more abrupt. When performing 1,000 iterations with the gateway, the average response time of Protocol Buffers increased by 17.72%, the throughput had a 14.55% decrease, and the median response time had a 2.15% increase. However, when performing 10,000 iterations with a gateway, the performance for Protocol Buffers surpassed that of the JSON serialization, the average response time decreased by 5.77%, the throughput had a 5.65% increase, and the median response time had a 5.57% reduction.

Table 15 - Benchmark results for retrieving all product entities

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 18.23 | 54.01 | 6.29 | 97.04 | 2.84 |
| Protocol Buffers | 1,000 | 16.97 | 57.97 | 5.88 | 533.22 | 2.75 |
| JSON | 10,000 | 4.67 | 202.82 | 2.57 | 77.20 | 1.07 |
| Protocol Buffers | 10,000 | 4.31 | 219.50 | 2.29 | 64.82 | 1.06 |
| JSON + Gateway | 1,000 | 21.62 | 45.64 | 9.64 | 213.46 | 5.13 |
| Protocol Buffers + Gateway | 1,000 | 19.63 | 50.19 | 7.70 | 354.09 | 4.12 |
| JSON + Gateway | 10,000 | 5.53 | 172.59 | 4.00 | 72.98 | 2.05 |
| Protocol Buffers + Gateway | 10,000 | 5.06 | 187.44 | 3.64 | 76.45 | 1.89 |

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 iterations test shows a 6.91% average response time reduction on Protocol Buffers, with an increase in throughput of 7.33% and a decrease in the median response time of 6.62%. Moreover, when increasing the number of iterations performed, the difference in performance also increases slightly, in this case, making the average response time decrease about 7.71%, increasing the throughput by 8.22% and decreasing the median response time by 10.89%.

Finally, when analyzing 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 iterations, it registered a 20.12% decrease in terms of the median response time and a 9.2% average response time reduction, and the throughput is also 9.96% higher than its counterpart. Even in the 10,000 iterations tests with a gateway, the same pattern can be observed, even if it diminished slightly.

Table 16 - Benchmark results for updating a user

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 5.05 | 184.89 | 3.85 | 66.14 | 2.23 |
| Protocol Buffers | 1,000 | 5.06 | 176.93 | 3.83 | 68.79 | 2.40 |
| JSON | 10,000 | 3.71 | 246.50 | 3.29 | 64.14 | 1.93 |
| Protocol Buffers | 10,000 | 3.89 | 221.85 | 3.67 | 48.64 | 2.07 |
| JSON + Gateway | 1,000 | 6.49 | 145.59 | 5.21 | 131.51 | 3.55 |
| Protocol Buffers + Gateway | 1,000 | 9.61 | 98.24 | 8.39 | 74.04 | 6.65 |
| JSON + Gateway | 10,000 | 5.12 | 181.73 | 4.85 | 65.08 | 3.04 |
| Protocol Buffers + Gateway | 10,000 | 7.81 | 119.39 | 7.37 | 113.64 | 5.64 |

Table 16 shows the data from the user data update, it is the second least data-intensive endpoint out of the 6 endpoints being tested. In general, it shows that Protocol Buffers are inferior to JSON in every test performed. The first test case without a gateway, when looking at the average response size, the difference is of only 10 microseconds, and in the median, the difference is about 20 microseconds, with the throughput having a more pronounced difference, with a 4.30% increase in throughput in JSON.

Furthermore, the tests performed demonstrate a significant disparity between Protocol Buffers and JSON, in favour of JSON. For 10,000 requests, Protocol Buffers had a 4.85% increase in the average response time, with the median response time also having an 11.55% increase and the throughput registering a 10% reduction.

Finally, when adding the gateway, either in the 1,000 iterations and in the 10,000 iterations, the performance of Protocol Buffers took a beat, in the 1,000 iterations, the average response time increased by about 48.07% and the median response time increased by 61.04% and the throughput also presented a 32.52% decrease. The same phenomenon happens when performing 10,000 iterations, the average response time increased by about 52.54%, the median response time increased by 51.96%, and the throughput had a 34.30% reduction.

Table 17 - Benchmark results for deleting a user

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 1,000 | 3.96 | 239.03 | 2.99 | 65.91 | 1.82 |
| Protocol Buffers | 1,000 | 4.86 | 195.48 | 3.60 | 60.83 | 2.19 |
| JSON | 10,000 | 2.54 | 361.82 | 2.26 | 62.19 | 1.02 |
| Protocol Buffers | 10,000 | 2.57 | 357.67 | 2.26 | 43.77 | 0.88 |
| JSON + Gateway | 1,000 | 5.91 | 161.91 | 4.90 | 67.76 | 3.17 |
| Protocol Buffers + Gateway | 1,000 | 5.54 | 172.52 | 4.59 | 58.83 | 2.95 |
| JSON + Gateway | 10,000 | 4.07 | 230.38 | 3.84 | 64.66 | 1.86 |
| Protocol Buffers + Gateway | 10,000 | 4.04 | 232.00 | 3.83 | 63.03 | 1.89 |

Table 17 shows the data of the deletion of a user, representing the endpoint with the least required serialization to be performed. Starting off with the first test case which were performed 1,000 iterations without a gateway, it has quite a pronounced disparity in performance, resulting in Protocol Buffers having worse performance than JSON. Upon making a comprehensive analysis of the data, it is revealed that the average response time has a 22.73% increase in seconds, there is also a 20.40% increase in the median response time, and a 18.21% decrease in throughput.

Furthermore, in the subsequent tests, the discrepancy between Protocol Buffers and JSON is marginally reduced when compared to the initial test case executed. For instance, in the second test, where 10,000 iterations were performed and without the gateway, the difference in the average response time is 30 microseconds in favour of JSON, with the median remaining consistent in both cases, the throughput exhibited a modest improvement for JSON, but nothing considerable.

Finally, when anlysing both the tests where the gateway was added, it is possible to see that in the 1,000 iterations test case, Protocol Buffers exhibited marginal superiority, in terms of the average response time, it decreased 6.26%, and in the median it decreased 6.33%, having the throughput improved 6.55%. However, when performing 10,000 iterations with a gateway, this difference in performance diminishes quite significantly, having the average response time being just 30 microseconds less than JSON, the median is also just 10 microseconds decrease, in favour of Protocol Buffers and the throughput also being just more or less a 2 requests per second increase.

#### Hypothesis Tests

In order to further validate the obtained analysis, hypothesis tests were performed. These are statistical methods used to validate our assumption hypothesis. For this task, Python was used with the SciPy [48] library, which offers all the algorithms for the 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, this is because each endpoint has its 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 Shapiro-wilk[49], as such, a test that doesn’t view the small deviations is important, the one chosen was D’Agostinho and Pearson’s [50], for two main reasons, the first one being that the SciPy library uses that one as the default for the normality test [51], 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 [52], 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 [53], 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 |
| Update user | PUT | Protocol Buffers have a lower response time |
| Delete user | DELETE | Protocol Buffers have a lower response time |
| Get all product entities | GET | Protocol Buffers have a lower response time |
| Get all users | GET | Protocol Buffers have a lower response time |
| Get user by ID | GET | Protocol Buffers have a lower response time |

For all endpoints, the three repeated trials for each serialization format were aggregated. Code Snippet 17 the base structure to perform these tests. The code and the results can all be found at /Data\_Analysis directory. It should be noted that the following code snippet involves looping through the data, which results in certain variables not being present within the shown code snippet.

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

The results of the hypothesis tests for the response time are as follows. The statistical significance observed in the user creation process across all tests performed, both with and without the gateway, aligns 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 serialization 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 three out of the four tests. The three instances where statistical significance is absent are observed in the 1,000 and 10,000 iterations without a gateway and the 1,000 iterations with the gateway. The sole test that demonstrates statistical significance is the final test with 10,000 iterations with a gateway. This behavior was also previously analyzed, as the more requests were performed, the closer the Protocol Buffers were to the performance of JSON and even surpassed it in the last test. In this case, the only endpoint accepting the alternative hypothesis is the test where 10,000 iterations were performed with a gateway. However, the other three tests accept the null hypothesis (H0), which states that there is no difference in response time between Protocol Buffers and 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 furthermore, it’s also a serialization-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 median of the first test case when performing 1,000 iterations without a gateway, but even there, the difference was marginal. This is also one of the endpoints for which the amount of data required for serialization is minimal. In addition, the null hypothesis was confirmed in all instances following the execution of the designated tests at this particular 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 1,000 iterations with a gateway. This is the endpoint with the least serialization out of the six endpoints tested. Like the retrieval of user information by its id, it has one test which accepts the alternative hypothesis, in this case, the test with 1,000 iterations with a gateway, while the rest of the tests accept the null hypothesis.

### 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 2.4.2. Furthermore, Kepler is a Prometheus exporter, with this we can connect with Grafana and use their pre-made dashboard[54], 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.

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 utilization of either JSON or Protocol Buffers, with or without a gateway. The tables further enumerate the number of requests executed for that test, the total wasted energy in joules for that test, the average and median energy consumption in joules, and finally, the energy consumption per request.

Table 19 - Energy consumption of user creation in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 456.95 | 4.71 | 4.36 | 0.46 |
| Protocol Buffers | 1,000 | 426.72 | 4.54 | 4.58 | 0.43 |
| JSON | 10,000 | 5111.80 | 5.51 | 5.62 | 0.51 |
| Protocol Buffers | 10,000 | 5117.89 | 5.54 | 5.67 | 0.51 |
| JSON + Gateway | 1,000 | 611.41 | 6.11 | 7.00 | 0.61 |
| Protocol Buffers + Gateway | 1,000 | 552.67 | 5.82 | 6.71 | 0.55 |
| JSON + Gateway | 10,000 | 6418.53 | 6.84 | 6.97 | 0.64 |
| Protocol Buffers + Gateway | 10,000 | 6104.71 | 6.62 | 6.73 | 0.61 |

Table 19 shows the gathered energy data for creating a user endpoint. Upon analysing that data, there is only one situation where the energy consumption of Protocol Buffers is worse than JSON, and that is the test case with 10,000 requests without a gateway. Even though the differences are marginal, it’s still an important consideration.

Moreover, the initial test case in which 1,000 requests were executed without a gateway revealed that Protocol Buffers exhibited a 6.62% reduction in energy expenditure when compared with JSON. With regard to the average energy consumption, this test demonstrated that Protocol Buffers consumed 3.61% less energy than JSON, in the case of the median, which exhibited a 5.05% increase in median energy consumption.

Finally, the results of the final two tests with the gateway demonstrate that Protocol Buffers exhibited a reduced energy consumption in comparison to JSON. However, it is noteworthy that as the number of requests increases, the disparity in energy expenditure between the two protocols diminishes. In this instance, when a total of 1,000 requests were executed with a gateway, Protocol Buffers demonstrated a cumulative energy expenditure reduction of 9.61%, accompanied by an average energy consumption decrease of 4.75% and a median energy consumption reduction of 4.14%. In the subsequent experiment, involving 10,000 requests, a decline in the discrepancy between the mean values was observed. The mean energy consumption exhibited a 4.89% decrease, while the median and mean values of JSON remained comparatively lower, with the average discrepancy diminished to 3.22%, and the median to 3.44%.

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 21.83 | 3.12 | 3.12 | 0.02 |
| Protocol Buffers | 1,000 | 25.26 | 3.61 | 3.61 | 0.03 |
| JSON | 10,000 | 94.61 | 3.15 | 2.32 | 0.01 |
| Protocol Buffers | 10,000 | 68.91 | 2.55 | 2.02 | 0.01 |
| JSON + Gateway | 1,000 | 49.57 | 7.08 | 7.08 | 0.05 |
| Protocol Buffers + Gateway | 1,000 | 50.16 | 7.17 | 7.17 | 0.05 |
| JSON + Gateway | 10,000 | 205.16 | 4.88 | 5.26 | 0.02 |
| Protocol Buffers + Gateway | 10,000 | 170.84 | 4.27 | 4.64 | 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 a different pattern to the previously analysed table. In this case, for both the test cases where 1,000 requests were made with or without a gateway, Protocol Buffers were inferior. In the first case, where the gateway was not used, the difference is quite steep, with increases of 15.71% in total, average and median energy consumption. The introduction of the gateway serves to reduce the discrepancy. In this instance, there is an increase in total energy consumption of 1.19%, with average and median energy consumption increasing by 1.27%. It is important to note that the dataset for the test cases with 1,000 requests is small due to the metric being the energy consumption per second. This means that the less time the test takes, the less data it will have, making the mean and average close to each other.

Furthermore, when increasing the number of requests, a different pattern emerges. In this case, there is an overall reduction of energy consumption. For instance, in the test case with 10,000 requests without a gateway, there is a total energy consumption reduction of 27.16%, with an average energy consumption reduction of 19.05%, and the median energy consumption also got a reduction of 12.93%. A similar trend was observed when using a gateway, where a 16.73% decrease in total energy consumption, a 12.5% decrease in average energy consumption, and an 11.79% decrease in median energy consumption were recorded.

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 40.07 | 6.68 | 6.68 | 0.04 |
| Protocol Buffers | 1,000 | 32.80 | 5.47 | 6.20 | 0.03 |
| JSON | 10,000 | 35.93 | 1.80 | 1.80 | 0.01< |
| Protocol Buffers | 10,000 | 38.97 | 1.86 | 1.87 | 0.01< |
| JSON + Gateway | 1,000 | 52.15 | 7.45 | 7.45 | 0.05 |
| Protocol Buffers + Gateway | 1,000 | 53.26 | 7.61 | 7.61 | 0.05 |
| JSON + Gateway | 10,000 | 159.57 | 4.56 | 5.37 | 0.02 |
| Protocol Buffers + Gateway | 10,000 | 137.58 | 4.17 | 4.45 | 0.01 |

Table 21 shows the gathered data for the energy consumption of the retrieval of a user by its id. The initial test case, involving 1,000 requests without a gateway, demonstrated that Protocol Buffers exhibited a reduced energy consumption, with an 18.14% total energy consumption reduction, an 18.11% reduction in the average energy consumption, and a 7.19% reduction in the median energy consumption.

Furthermore, on the test where 10,000 requests were performed without a gateway, the energy consumption increased on the Protocol Buffers side, with the total energy consumption increasing by 8.46% and the average energy consumption increasing by 3.33%, and the median energy consumption also increasing by 3.89%. The aforementioned behaviour is also observed in the context of testing for 1,000 requests with a gateway, wherein the data reveals a 2.13% increase in total energy consumption and a 2.15% increase in the average and median energy consumption.

Finally, when performing the test with 10,000 requests with a gateway, the energy consumption decreased significantly, for instance, the total energy consumption decreased by 13.78%, while the average energy consumption decreased by 8.55%, and the median energy consumed decreased by 17.13%

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 27.58 | 1.38 | 1.51 | 0.03 |
| Protocol Buffers | 1,000 | 20.07 | 1.12 | 1.12 | 0.02 |
| JSON | 10,000 | 200.51 | 4.01 | 4.33 | 0.02 |
| Protocol Buffers | 10,000 | 167.53 | 3.56 | 3.39 | 0.02 |
| JSON + Gateway | 1,000 | 51.26 | 2.23 | 2.23 | 0.05 |
| Protocol Buffers + Gateway | 1,000 | 82.94 | 3.95 | 3.87 | 0.08 |
| JSON + Gateway | 10,000 | 434.77 | 7.37 | 4.58 | 0.04 |
| Protocol Buffers + Gateway | 10,000 | 309.22 | 5.62 | 2.94 | 0.03 |

As illustrated in, Table 22, the data collated pertains to the energy consumption of the retrieval of all product entities. The initial two test cases demonstrated a favourable representation of Protocol Buffers energy expenditure. For the 1,000 requests without a gateway, the total energy expenditure saw a 27.23% decrease, with the average energy consumption also having an 18.84% decrease, and a 25.83% decrease on the median energy consumption. In the test where 10,000 requests were made without a gateway, results remained consistent, with a 16.45% decrease in total energy consumption, a 11.22% decrease in average energy consumption, and a 21.71% decrease in median energy expenditure.

In the subsequent analysis of the test involving 1,000 requests to a gateway, an anomalous outcome was observed. The energy consumption showed an abrupt increase, indicating that Protocol Buffers consumed almost twice the energy of JSON across all metrics. Further analysis of the discrepancies revealed the origin of the issue. In this instance, the gateway was identified as the source of the problem.

An analysis of the total energy consumption of the gateway revealed that the JSON usage required approximately 24.85 joules, whereas the Protocol Buffer usage demanded a significantly higher energy expenditure of 59.58 joules. This substantial discrepancy in energy usage had a notable impact on the overall results, particularly in the context of the summed values. In the absence of a gateway, the energy expenditure of Protocol Buffers was found to be marginally lower compared to that of JSON.

Finally, in the last test, where 10,000 requests were made with the gateway, a 28.88% decrease in total energy consumption was observed. Furthermore, the average energy consumption decreased by 23.74%, while the median decreased by 35.81%.

Table 23 - Energy consumption of updating a user in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 11.95 | 1.99 | 1.99 | 0.01 |
| Protocol Buffers | 1,000 | 36.64 | 5.23 | 5.23 | 0.04 |
| JSON | 10,000 | 127.92 | 3.05 | 1.97 | 0.01 |
| Protocol Buffers | 10,000 | 128.48 | 2.73 | 1.83 | 0.01 |
| JSON + Gateway | 1,000 | 24.31 | 3.47 | 3.47 | 0.02 |
| Protocol Buffers + Gateway | 1,000 | 54.97 | 5.00 | 5.00 | 0.05 |
| JSON + Gateway | 10,000 | 317.71 | 5.67 | 6.68 | 0.03 |
| Protocol Buffers + Gateway | 10,000 | 555.01 | 6.53 | 7.21 | 0.06 |

As shown in Table 23, the energy consumption of the test designed to update the user information is documented. This evaluation constitutes a significant challenge for Protocol Buffers. Most of the test cases in this endpoint reveal that Protocol Buffers increases total energy consumption by double the amount when compared with JSON. To illustrate the point, consider the initial scenario in which 1,000 requests were executed without the utilisation of a gateway. The disparity in performance metrics is pronounced.

Despite the demonstrable superiority of JSON in terms of total energy consumption in this test, a more detailed analysis of the second test, which involved 10,000 requests without a gateway, reveals that Protocol Buffers, in fact, consumed a higher total amount of energy. However, when the average and median values are considered, in this test case scenario, Protocol Buffers uses less energy than JSON.

Table 24 - Energy consumption of deleting a user in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Requests | Total Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1,000 | 16.71 | 3.34 | 3.34 | 0.02 |
| Protocol Buffers | 1,000 | 17.17 | 2.86 | 2.86 | 0.02 |
| JSON | 10,000 | 67.47 | 2.33 | 1.74 | 0.01 |
| Protocol Buffers | 10,000 | 67.85 | 2.34 | 1.89 | 0.01 |
| JSON + Gateway | 1,000 | 31.44 | 4.49 | 3.70 | 0.03 |
| Protocol Buffers + Gateway | 1,000 | 50.54 | 7.22 | 8.91 | 0.05 |
| JSON + Gateway | 10,000 | 232.31 | 5.28 | 3.70 | 0.02 |
| Protocol Buffers + Gateway | 10,000 | 209.45 | 4.76 | 3.35 | 0.02 |

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 in total, Protocol Buffers expended 2.75% more in total. However, the average and the median energy expenditure were both 14.37% lower than JSON. If Protocol Buffers had been slightly more performant than JSON, it would have resulted in a marginal reduction in energy expenditure, which can be observed through Table 17, where Protocol Buffers are slower by almost 1 millisecond on average.

The rest of the results were worse for Protocol Buffers in general. In the 10,000 requests without a gateway, the difference in energy consumption is marginal, but still worse than JSON. For the 1,000 requests with a gateway the differences were abysmal, for the total energy expenditure, Protocol Buffers wasted 60.75% more energy than its counterpart. For the last test where 10,000 requests were made with a gateway, the total energy expenditure of Protocol Buffers was 74.69% more than JSON, but the average and median differences were slightly less than that, with the average energy consumption being 15.17% more and the median energy consumption being 7.93%.

#### Hypothesis Tests

To further perform validations, hypothesis tests were also performed for the energy consumption. This allows statistical validation of the observed data on energy consumption, the same process explained in section 8.2.3.1 was also followed in this section, first test for normality using the best-suited algorithm, in this case, as there are datasets smaller than 100, Shapiro-Wilk test might be used, but if the dataset exceeds 100 then the D’Agostinho and Pearson’s algorithm is used, and after that choose between doing a T-test if it follows a normal distribution or Mann-Whitney U if it doesn’t follow a normal distribution.

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 | Protocol Buffers are more energy efficient than JSON |
| Get all users | GET | Protocol Buffers are more energy efficient than JSON |
| Get user by ID | GET | Protocol Buffers are more energy efficient than JSON |

As illustrated in Table 25, the hypothesis tests being conducted are outlined, and the code structure aligns with code snippet 5, which was previously discussed. To verify the code, the reader is instructed to proceed to the Data\_analysis directory and open the Hypothesis\_test.ipynb file.

The results of the hypothesis tests are as follows. For the initial endpoint, the tests conducted without the gateway demonstrated no statistically significant difference, thereby validating the null hypothesis. This outcome indicates that the energy efficiency of Protocol Buffers is comparable to that of JSON, with no significant variation observed in the absence of a gateway. In contrast, upon incorporating the gateway, both outcomes prove to be statistically significant, thus validating the alternative hypothesis, which indicates that Protocol Buffers demonstrate superior energy efficiency in comparison to JSON.

For the retrieval of all users, only the tests where the 1,000 requests with or without the gateway stated that there was no statistical significance, and thus accepting the null hypothesis. However, when performing the 10,000 requests with or without the gateway, the tests stated that there was a significant statistical difference to accept the alternative hypothesis.

For the subsequent endpoint, characterized by the retrieval of user information by its id, the hypothesis tests demonstrated that the test case involving 10,000 requests made without a gateway, as well as the test case involving 1,000 requests made with a gateway, showed no statistical significance, this outcome serves to accept the null hypothesis. However, the results of the test in which 1,000 requests were made without a gateway, as well as the test in which 10,000 requests were made with a gateway, indicated a statistical significance. This permitted the acceptance of the alternative hypothesis for these two tests.

For the endpoint to retrieve all product entities, it was found that only one of the tests stated that there was no statistical significance. This was the test with 1,000 requests with a gateway, which ended up accepting the null hypothesis. Nevertheless, the remaining tests were found to be statistically significant, thus validating the alternative hypothesis.

The endpoint for updating a user has only one hypothesis test that is stated to be statistically significant, that test being the one with 1,000 requests with a gateway, this one accepted the alternate hypothesis. The rest, on the other hand, accept the null hypothesis.

Finally, concerning the final endpoint, the deletion of a user, it was determined that the tests that did not demonstrate statistical significance were those with 10,000 requests without a gateway and the one with 1,000 requests with a gateway. In both cases, the null hypothesis was accepted. However, the experiment in which 1,000 requests were made without a gateway, and the experiment in which 10,000 requests were made with a gateway, are statistically significant. This indicates that the alternate hypothesis, namely that the Protocol buffers are more energy efficient than JSON, is accepted.

## Conclusion

The subsequent chapter 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 exhibited marginal superiority over JSON in three out of four of the evaluated endpoints. The sole endpoint where Protocol Buffers was not faster was the creation of a user, a process that is more computationally intensive and can potentially impede serialization. For the remaining three endpoints, Protocol Buffers demonstrated a clear advantage in terms of performance over JSON. However, it should be noted that the average execution time for Protocol Buffers could potentially be higher. A more thorough examination of the data, incorporating metrics such as the median and throughput, revealed a contrasting outcome. This discrepancy can be attributed to the presence of a higher number of outliers, thereby substantiating the hypothesis that Protocol Buffers exhibit superior performance compared to JSON, if the right metrics are analysed.

Secondly, with respect to energy efficiency, Protocol Buffers demonstrated variable performance, exhibiting both superior and inferior outcomes in different scenarios. Two instances exhibited reduced energy consumption, while the remaining two endpoints exhibited increased energy expenditure. However, a salient point is the total energy consumption, which showed that Protocol Buffers consumed less overall than JSON in 3 of the test cases, the only one that wasted more energy was retrieving a user by its id. This scenario required further investigation, leading to the conclusion that as the size of the serialized data increases, Protocol Buffers exhibit enhanced performance and demonstrate a potential correlation between performance and energy consumption. The analysis revealed that the faster Protocol Buffers were, the less energy was consumed. This phenomenon was observed during the retrieval of all users and product entities. While Protocol Buffers exhibited faster retrieval speeds, they also consumed more energy on average. However, when considering the total energy expenditure, Protocol Buffers were found to be more energy efficient.

In conclusion, Protocol Buffers have been shown to exhibit superior performance in scenarios involving high serialization intensity when compared to JSON. Furthermore, processes that are more demanding than serialization processes may potentially overshadow the gains in performance, resulting in diminishing returns. With respect to energy consumption, it was observed that Protocol Buffers exhibited 50% energy efficiency superiority over JSON in terms of average energy expenditure. However, when the total energy consumption was assessed, Protocol Buffers demonstrated a significant reduction in energy expenditure compared to JSON. It should be noted that this phenomenon is not universally applicable, in cases where the serialization process is less complex, Protocol Buffers has been observed to consume more energy than JSON. For a more detailed view of the processes that were run, refer to the repository [55] where all the data, documents, setup and project can be found.

# Conclusion

The subsequent chapter documents the accomplishments throughout the dissertation's development process, showcasing the outcomes of the conducted analyses and the insights derived from the study. Furthermore, it provides the challenges that were faced, encompassing difficulties related to knowledge and implementation. Finally, the chapter concludes with a discussion of threats to validity that could be posed by the study performed.

## 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 serialization format, and which, might be better or worse in terms of energy efficiency or performance. As anticipated, as the volume of data to be serialized increases, the performance and energy efficiency of Protocol Buffers compared to JSON improve.

## 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 facilitate the utilization 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 serialization format with which I had no prior experience. The complexity of debugging these tools contributed to the overall complexity of the implementation process.

## 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.

It is important to note the lack of, prior experience with Protocol Buffers introduced difficulties in comprehending the scope of acceptable practices, particularly in the context of analyzing performance and energy efficiency. This challenge can be more easily overcome by individuals with extensive experience with Protocol Buffers and possess the expertise to discern potential issues in the work being conducted.

.

References

[1] 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.

[2] 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

[3] 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.

[4] 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.

[5] D. Persson and N. Carlsson, “Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV,” 2020.

[6] 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.

[7] 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.

[8] M. Newton Hedelin, “Benchmarking and performance analysis of communication protocols : A comparative case study of gRPC, REST, and SOAP,” 2024, Accessed: Nov. 02, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-351588

[9] 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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] “Ordem dos Engenheiros Código de Ética e Deontologia”.

[19] R. T. Fielding, “Fielding dissertation : Chapter 5 : Representational state transfer (rest),” *http://www.ics.uci.edu/〜fielding/pubs/dissertation/rest\_arch\_style.htm*, 2000, Accessed: Oct. 14, 2024. [Online]. Available: https://cir.nii.ac.jp/crid/1572543025856338816

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

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

[22] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000.

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

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

[25] C. Currier, “Protocol Buffers,” *Mobile Forensics – The File Format Handbook*, pp. 223–260, 2022, doi: 10.1007/978-3-030-98467-0\_9.

[26] 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

[27] 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.

[28] 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

[29] 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.

[30] G. Bath, R. Black, A. Podelko, A. Pollner, and R. Rice, “Certified Tester Foundation Level Syllabus-Performance Testing,” 2018.

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

[32] 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.

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

[34] “Developing authentic charisma | LinkedIn Learning.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.linkedin.com/learning/nano-tips-for-developing-magnetic-charisma-with-shade-zahrai/developing-authentic-charisma

[35] K. Blanchard, “Welcome | LinkedIn Learning.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.linkedin.com/learning/ken-blanchard-on-servant-leadership/welcome

[36] B. McGannon, “Project risk: A battle of resiliency | LinkedIn Learning.” Accessed: Dec. 24, 2024. [Online]. Available: https://www.linkedin.com/learning/project-management-foundations-risk-14910484/project-risk-a-battle-of-resiliency

[37] 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.

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

[39] 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.

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

[41] “1230199/control\_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/control\_project/general

[42] “1230199/experimental\_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/experimental\_project/general

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

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

[45] V. R. Basili, G. Caldiera, and H. D. Rombach, “THE GOAL QUESTION METRIC APPROACH”.

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

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

[48] “SciPy.” Accessed: Mar. 20, 2025. [Online]. Available: https://scipy.org/

[49] “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

[50] “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

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

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

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

[54] “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

[55] “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

A screenshot of a computer

Description automatically generatedAppendix A

Figure 15 - Work Breakdown Structure

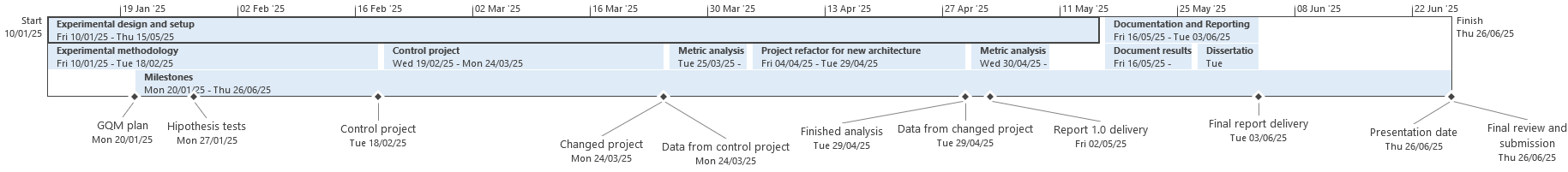


Figure 16 - Full timeline

A close-up of a computer screen

Description automatically generated

Figure 17 - Improved resolution of timeline part 1/2

A screenshot of a computer

Description automatically generated

Figure 18 - Improved resolution of timeline part 2/2

A screenshot of a computer screen

Description automatically generated

Figure 19 - Risk Register Part 1

A screenshot of a computer

Description automatically generated

Figure 20 - Risk Register Part 2

Appendix B

| **Publication** | **Year** | **Conclusion** |
| --- | --- | --- |
| Impacts of data interchange formats on energy consumption and performance in smartphones | 2011 | JSON > Protocol Buffers > Protocol Buffers with GZIP > JSON with GZIP |
| Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication | 2011 | JSON > Protocol Buffers |
| Performance evaluation of object serialization libraries in XML, JSON and binary formats | 2012 | JSON>Apache Thrift > Protocol Buffers> Apache Avro |
| Google protocol buffers research and application in online game | 2013 | JSON > Protocol Buffers |
| Integrating a system for symbol program  ming of real processes with a cloud service | 2015 | JSON>MessagePack |
| Performance evaluation of using Protocol Buffers in the Internet of Things communication | 2016 | **In most cases**: JSON > BSON > Protocol Buffers. However, **in some cases**: BSON> JSON > Protocol Buffers |
| Smart grid serialization comparison: Comparison of serialization for distributed control in the context of the Internet of Things | 2017 | BSON > CBOR > JSON > BSON with GZIP > Smile > Protocol Buffers > CBOR with GZIP > JSON with GZIP > Apache Avro > Protocol Buffers with GZIP > Smile with GZIP > MessagePack > Apache Avro with GZIP > MessagePack with GZIP |
| Binary Representation of Device Descriptions: CBOR versus RDF HDT | 2018 | JSON > CBOR |
| Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs | 2018 | FlatBuffers > Protocol Buffers > MessagePack |
| Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats | 2018 | JSON > MessagePack > Protocol Buffers > Apache Avro |
| Analytical assessment of binary data serialization techniques in IoT context (evaluating protocol buffers, flat buffers, MessagePack, and BSON for sensor nodes) | 2019 | For numeric and mixed data: BSON >FlatBuffers > MessagePack > Protocol Buffers. For textual data: FlatBuffers > BSON > MessagePack > Protocol Buffers |
| Enabling Model-Driven Software Development Tools for the Internet of Things | 2019 | JSON > FlatBuffers |
| Flatbuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format | 2019 | JSON > FlatBuffers |
| Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV | 2020 | JSON > FlatBuffers > Protocol Buffers |

Table 26 - “A list of space-efﬁciency benchmark publications that involve JSON[…]”[2]

Adapted from [2]

Appendix D

The following images 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 18 - 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 19 - 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 20 - 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 21 - UserRoles generated proto files

As illustrated in Code Snippets 18 to 21, 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 the explicit indication to use the proto3 syntax.

Furthermore, we have the following 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 22 - 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 23 - 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 24 - ProductEntityDTO generated proto file

As with the User domain, the Product Entity domain generated proto files are visible in Code Snippets 22 to 24. 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.