

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

**Miguel Alves Ferreira**

**Dissertação**

**Engenharia Informática**

**Área de Especialização em Engenharia de software**

Integrity Statement

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

I have not plagiarized 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, 14 de abril de 2025

Summary

O

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

Abstract

The .

**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 essentially 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 why I know how to program, and for that I am very grateful. I'd 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 2

1.3 Objectives and Research Questions 2

1.4 Ethical Considerations 3

1.5 Report Structure 3

2 Background 4

2.1 REST 4

2.1.1 How are REST Applications Structured 6

2.2 JSON 7

2.3 Protocol Buffers 9

2.3.1 Structure of Protocol Buffers 9

2.3.2 Schema Compilation and Code Generation 10

2.4 Performance and Energy Analysis Tools 11

2.4.1 Grafana K6 11

2.4.2 Kepler 11

3 Planning 13

3.1 Project Charter 13

3.1.1 Stakeholders 13

3.1.2 Scope 14

3.1.3 Objectives 14

3.1.4 Benefits 15

3.1.5 Deliverables 15

3.1.6 Time 15

3.1.7 Costs 16

3.1.8 Assumptions and Restrictions 16

3.1.9 Risk 16

3.2 Work Breakdown Structure 17

3.3 Work Breakdown Structure Dictionary 17

3.4 Timeline 21

3.5 Skills 22

3.5.1 Required Skills 22

3.5.2 Strengths 22

3.5.3 Weaknesses 23

3.5.4 Technical Skills 23

3.5.5 Plans for Improvement 25

4 Literature Review 26

4.1 Research Questions 26

4.2 Data Sources 26

4.3 Search Terms 27

4.4 Eligibility Criteria 28

4.5 Data Collection Process 28

4.6 Discussion 29

4.6.1 RQ1: How does using ProtoBufs impact the performance of HTTP-based REST architectures? 29

4.6.2 RQ2: To what extent can ProtoBufs improve energy efficiency in HTTP-based REST architecture communications compared to JSON? 34

4.7 Conclusion 36

5 Research Methodology 37

5.1 Why Controlled Experiment Methodology? 37

5.2 Data Collection 37

6 Analysis and Migration 38

6.1 Project 38

6.1.1 Context 38

6.1.2 Changes Made to Base Project 39

6.2 Migration 39

6.2.1 Components 39

6.2.2 Strategy 40

7 Implementation 41

7.1 Dependencies and Plugins 41

7.2 User Domain Migration 44

7.3 Product Entity Domain Migration 46

7.4 Kubernetes Cluster Setup 47

8 Experiment 50

8.1 Goal Question Metric 50

8.1.1 Performance 51

8.1.2 Energy Consumption 51

8.2 Performed Experiments 52

8.2.1 Environment 52

8.2.2 Performance Tests Setup 52

8.2.3 Performance Tests 53

8.2.4 Energy Consumption Tests Setup 57

8.2.5 Energy Consumption Tests 57

8.3 Conclusion 60

9 Conclusion 62

9.1 Accomplishments 62

9.2 Difficulties 62

9.3 Threats to Validity 63

List of Figures

[Figure 1 - REST API in action 5](#_Toc195546023)

[Figure 2 - Overall compliance with the REST principles of 500 REST APIS 6](#_Toc195546024)

[Figure 3 - Timeline part 1/2 21](#_Toc195546025)

[Figure 4 - Timeline part 2/2 22](#_Toc195546026)

[Figure 5 - Prisma systematic methodology 29](#_Toc195546027)

[Figure 6 - Average serialization time 31](#_Toc195546028)

[Figure 7 - Average deserialization time 31](#_Toc195546029)

[Figure 8 - Response time benchmark of a GET request uncompressed 32](#_Toc195546030)

[Figure 9 - Payload size benchmark with flat data 33](#_Toc195546031)

[Figure 10 - Memory analysis of JSON serialization 33](#_Toc195546032)

[Figure 11 - Memory analysis of Protocol Buffers serialization 34](#_Toc195546033)

[Figure 12 - Energy expended on CPU with data synchronization (volume1) 35](#_Toc195546034)

[Figure 13 - Energy expended on CPU with data synchronization (volume2) 35](#_Toc195546035)

[Figure 14 - Representation of possible GQM approach 50](#_Toc195546036)

[Figure 15 - Work Breakdown Structure 69](#_Toc195546037)

[Figure 16 - Full timeline 70](#_Toc195546038)

[Figure 17 - Improved resolution of timeline part 1/2 70](#_Toc195546039)

[Figure 18 - Improved resolution of timeline part 2/2 71](#_Toc195546040)

[Figure 19 - Risk Register Part 1 72](#_Toc195546041)

[Figure 20 - Risk Register Part 2 73](#_Toc195546042)

List of Tables

Table 1 - List of Types that can be defined in the *.proto* file 10

Table 2 - Power and interest matrix 13

Table 3 - WBS dictionary 18

Table 4 - Technical Skills needed to complete the dissertation 24

Table 5 - Data sources 27

Table 6 - Average serialization time in ms. Reprinted from [1] 30

Table 7 - Average deserialization time in ms. Reprinted from [1]. 30

Table 8 - Question and metrics for performance 51

Table 9 - Question and metrics for energy consumption 51

Table 10 - Testing hardware specification 52

Table 11 - Endpoints requested in the tests 53

Table 12 - Benchmark results for creating Users 54

Table 13 - Benchmark results for retrieving all Users 54

Table 14 - Benchmark results for retrieving User by id 54

Table 15 - Benchmark results for retrieving all Product Entities 54

Table 16 - Hypothesis tests for each endpoint in terms of performance 56

Table 17 - Energy consumption of User creation in joules 57

Table 18 - Energy consumption of retrieval of all Users in joules 58

Table 19 - Energy consumption of retrieval of User by its id in joules 58

Table 20 - Energy consumption of retrieval of all Product Entities in joules 58

Table 21 - Hypothesis tests for each endpoint in terms of energy consumption 59

Table 22 - “A list of space-efﬁciency benchmark publications that involve JSON[…]”[2] 76

List of Code Snippets

[Code Snippet 1 - Example of a User JSON 7](#_Toc195546138)

[Code Snippet 2 - Example of Decoding and Encoding a User JSON in Golang 8](#_Toc195546139)

[Code Snippet 3 - Block of code from a *.proto* file defining a User 9](#_Toc195546140)

[Code Snippet 4 - Search query developed with the search terms and research questions 27](#_Toc195546141)

[Code Snippet 5 - Changes to allow Protobufs in project 41](#_Toc195546142)

[Code Snippet 6 - Dependencies to compile Protobufs automatically 42](#_Toc195546143)

[Code Snippet 7 - User response DTO 43](#_Toc195546144)

[Code Snippet 8 - Generated user response Protobuf file 43](#_Toc195546145)

[Code Snippet 9 - Example of one endpoint that uses protobufs 44](#_Toc195546146)

[Code Snippet 10 - Example of service level insert method using Protobufs 45](#_Toc195546147)

[Code Snippet 11 - Helper methods to convert special data types into protobufs 46](#_Toc195546148)

[Code Snippet 12 - Portion of the control project to insert a new user at the service level 46](#_Toc195546149)

[Code Snippet 13 - Example of one endpoint for *Product Entity* domain 47](#_Toc195546150)

[Code Snippet 14 - Example of service level method to retrieve all *Product Entities* with protobufs 47](#_Toc195546151)

[Code Snippet 15 - Docker file for both projects 48](#_Toc195546152)

[Code Snippet 16 - Service monitor manifest file for API 48](#_Toc195546153)

[Code Snippet 17 - Deployment manifest file for API 49](#_Toc195546154)

[Code Snippet 18 - Response time hypothesis test structure 56](#_Toc195546155)

[Code Snippet 19 - UserDTO generated proto file 77](#_Toc195546156)

[Code Snippet 20 - EditUserDTO generated proto file 77](#_Toc195546157)

[Code Snippet 21 - LocalDateTimePb generated proto file 77](#_Toc195546158)

[Code Snippet 22 - UserRoles generated proto files 77](#_Toc195546159)

[Code Snippet 23 - ProductEntity generated proto file 78](#_Toc195546160)

[Code Snippet 24 - Product generated proto file 78](#_Toc195546161)

[Code Snippet 25 - ProductEntityDTO generated proto file 78](#_Toc195546162)

Acronyms and Symbols

**Acronym List**

**JSON** *Javascript Object Notation*

**IDL** *Interface Definition Language*

**XML** *Extensible Markup Language*

**REST** *Representational State Transfer*

**HATOAS** *Hypermedia As The Engine Of Application State*

**URI** *Uniform Resource Identifier*

**gRPC** *Google Remote Procedure Call*

**IoT** *Internet Of Things*

**HTTP** *Hypertext Transfer Protocol*

**API** *Application Programming Interface*

**Go** *Golang*

**Kepler** *Kubernetes Efficient Power Level Exporter*

**eBPF** *extended Berkeley Packet Filter*

# 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

In the field of computer engineering, it is essential to have efficient and scalable communication protocols so that they can respond to the needs presented by the system in which they are integrated. With The rise of cloud computing, IoT (Internet of Things), and microservices has led to a proliferation of REST (Representational State Transfer) APIs (Application Programming interfaces), which rely heavily on serialization formats like JSON (JavaScript Object Notation), XML (Extensible Markup Language), ProtoBufs (Protocol Buffers) and others, among these, JSON has emerged as the standard 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 of data [1], [2]. In contrast, ProtoBufs is a binary serialization format developed by Google that offers a more efficient, compact, and schema-driven alternative to JSON, not only that, but the 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, ProtoBuf's complexity and lack of human readability might limit its adoption in REST architectures, where JSON remains the dominant choice [6]. This study aims to evaluate the trade-offs between JSON and ProtoBufs in REST applications, focusing on performance, scalability, resource consumption, and energy efficiency to provide insights for developers and software architects seeking to optimize their communications in REST applications.

## Problem Description

Choosing the right technology for data serialization plays a major 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 serialization 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 gRPC (Google Remote Procedure Call), which is more performant than HTTP REST applications [8], [9]. Moreover, performance and energy efficiency are increasingly critical in today’s software development, particularly in HTTP-based REST architecture, this 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 serialization formats like ProtoBufs could reduce these bottlenecks by minimizing payload sizes and improving processing speed. Furthermore, energy efficiency is another growing concern, especially in environments like mobile devices, IoT, and data centers, 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 centers, serialization optimizations can lead to substantial energy savings, translating into lower costs and reduced environmental impact [12].

## Objectives and Research Questions

The Primary objective of the research is to analyze and compare the performance and energy efficiency of ProtoBufs and JSON in HTTP-based REST architectures, and, as such, we have the following research questions:

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

## 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 possible subjects, the study will firmly comply with the following ethical aspects:

1. Data Privacy and Security: All test data to be used in the study will be synthetic and generated specifically for benchmarking purposes. No real user data or proprietary information will be used, ensuring compliance with data protection regulations such as RGPD (Regulamento Geral de proteção de dados).
2. Transparency and Reproducibility: The experimental setup, tools, and methodologies are going to be documented in detail to allow reproducibility of the results by other researchers. Open-source tools and datasets will be used whenever possible [14].
3. Benchmarking Practices: There will be extreme care to avoid bias in performance evaluations. Both serialization formats will be tested under equal conditions, whenever possible, to ensure fair comparisons.

## Report Structure

# 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 serialization formats mentioned above, in this case, JSON and Protobuf.

## REST

Representational State Transfer (REST) is an architectural style introduced in 2000 [15]. 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 [16].

A diagram of a rest api

Description automatically generated

Figure 1 - REST API in action

Reprinted from [16]

At its core, REST leverages HTTP methods like GET, POST, PUT, DELETE, and PATCH to perform operations on resources. These methods correspond to common CRUD (Create, Read, Update, Delete) 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 serialized in various formats, such as JSON, XML, or ProtoBufs, 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 low 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 [17], [18]:

1. 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.
2. Client-Server Architecture: The client and server are separate entities, allowing them to evolve independently. The client is responsible for the user interface and request initiation, while the server handles data storage and business logic.
3. Uniform Interface: Rest applications must follow a standardized way of interacting with resources. This includes:

* Resource Identification, which is done through URIs (Uniform Resource Identifiers), like the following: */users/1 identifies* a specific user resource.
* Representation of resources is represented in formats like JSON, XML, or ProtoBufs.
* HATEOAS (Hypermedia as The Engine of Application State) is the principle that clients should navigate the application through hyperlinks embedded in responses. This allows for the dynamic discovery of resources and actions.

1. 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.
2. Cacheability: Responses from the server should explicitly indicate whether they are cacheable. The proper use of caching can reduce latency and bandwidth usage, improving overall performance.
3. Code on Demand: In essence, 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 (JavaScript Object Notation) was introduced in 2001 and has become one of the most widely used data serialization 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 [19].

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 handle JSON, thus allowing developers to effortlessly serialize and deserialize data. For example, in Go (Golang), JSON encoding and decoding are facilitated by the encoding/json package. For the following example how the encoding and decoding of JSON data can be achieved in Go (Golang) 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 Golang

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 ProtoBufs, were initially developed internally at Google in 2001 and released to the public in 2008. According to the official documentation, "Protocol buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data" [20]. ProtoBufs is fundamentally a binary serialization format that facilitates efficient and compact data exchange between applications. Its design prioritizes performance, compactness, and simplicity, making it an ideal choice for scenarios where high throughput and low latency are critical [5], [21], [22], [23].

ProtoBufs are extensively used in applications such as microservices, distributed systems, and Internet of Things (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 ProtoBufs for serializing and deserializing messages. This combination enables developers to build scalable and efficient APIs and distributed systems[8], [24]. While ProtoBufs are often noted for outperforming other serialization formats like XML and JSON in terms of speed and data compactness, comparisons must consider architectural differences, for instance, evaluations frequently compare ProtoBufs in gRPC-based architectures against JSON in RESTful architectures[4], [24], [25]. 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 serialization approach, meaning the data structure must be explicitly defined in a schema file with the *.proto* extension. This schema, written in ProtoBufs 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, 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.

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

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 ProtoBufs 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 ProtoBufs messages. For example, in Go (Golang), 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 utilization of software systems. These tools allow us to observe applications under diverse workloads and operational conditions. In this dissertation, two analytical tools are employed: k6 and 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 analyze system behaviour under variable load conditions. It has support for multiple protocols, including HTTP, HTTPS, and FTP, which enhances its utility as a comprehensive performance-testing solution.

**Main Types of Tests:**

1. **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.” [26]
2. **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.” [26]
3. **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.” [26]
4. **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.” [26]

### Kepler

Kepler (Kubernetes Efficient Power Level Exporter) is an open-source tool designed to monitor and estimate the energy consumption of Kubernetes workloads. By leveraging technologies like eBPF (extended Berkeley Packet Filter) 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 [27], [28]:

* **Data collection via eBPF and Hardware Counters:** This allows Kepler to directly collect performance metrics from the Linux kernel, and due to utilizing hardware counters, it is also capable of gathering detailed information on energy consumption estimates [27], [28].
* **Real-time power consumption metrics:** It can access data from hardware components through their API to get accurate and real-time power metrics [27], [28].
* **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 utilization 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 various 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 serialization 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

Nowadays in software engineering, the efficiency of data serialization has a direct influence on the performance of REST architectures, JSON, as the dominant serialization 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 (ProtoBufs) is a schema-driven binary serialization format, that offers significant performance advantages like reduced payload sizes, improved parsing speeds, and optimized resource consumption. Despite these benefits, this serialization format has increased complexity and isn’t human-readable, and as such, it creates a barrier to its adoption.

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

### Objectives

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**: Analyze the latency and throughput for both formats.
* **Scalability**: See how well each format performs under different loads and data sizes.
* **Energy Consumption**: View the CPU energy utilization during serialization and deserialization.
* **Resource Efficiency**: Compare memory utilization and processing overhead.

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

### Benefits

The benefits the current research will provide are:

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

### Deliverables

The project aims to produce the following deliverables:

1. **Project Plan:** Detailed timeline and task breakdown.
2. **Dissertation Report:** A thesis that documents the study, techniques, findings, and interpretations.
3. **Software Code:** All Code related to the experimental setup, including benchmarks and test scripts.
4. **Testing Reports:** Detailed analysis of performance metrics such as latency, throughput, and energy consumption.
5. **Data Collected:** Processed and raw data from benchmark experiments.
6. **Presentation and Discussion:** A final presentation summarizing 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 probably will change due to my lack of knowledge of the processes, but the mandatory dates will hopefully stay the same throughout the project.

**Milestones:**

* GQM (Goal Question Metric) – 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 and Restrictions

**Assumptions:**

1. Benchmarking tools and required libraries will be stable and accessible throughout the project.
2. Synthetic data will adequately simulate real-world scenarios.
3. Experimental environments will be consistent across tests.
4. Digital libraries for literature review will remain accessible.

**Restrictions**

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

### Risk

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 (Figures 18 and 19 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, resulting in a prioritization based on the PI score. Below are some examples:

* **High-Priority Risks**:
  + Synthetic data bias (PI Score: 15)
  + Time constraints limiting experimental scope (PI Score: 15)
* **Medium-Priority Risks**:
  + Tool incompatibility (PI Score: 8)
  + Data loss due to accidental corruption (PI 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 3 main phases. The experimental Design and Setup involves defining the research methodology, establishing the experimental environment, and collecting baseline data. It sets the foundation for the comparison between JSON and ProtoBufs, the Documentation and Reporting phase, and data from the experiments are analyzed 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 WBS (work breakdown structure) 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 mainly 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 analyzed – 15%  Define measurement methods -10%  Create GQM plan – 20%  Identify tools and technologies – 5%  Define Hypothesis – 15%  Acceptance from advisor – 30% |
| 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 analyzed and the measurement methods should also be defined. | Independent variables – 25%  Dependent variables – 25%  Define Metrics – 25%  Create a standardized 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 control project, | Acquire control project – 30%  Refactor needed parts – 10%  Setup monitoring tools – 30%  Setup 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 new 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 analyzed and documented, and an improved version of the dissertation is going to be written. | Analyze 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. | Analyze 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 that I am good at. 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 ProtoBufs is also something that I need to have a better understanding. The following table displays the technical skills needed to complete successfully the dissertation.

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 ProtoBufs 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 extract 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 ProtoBufs, 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. |
| ProtoBufs | 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. |
| JMeter | 9 | 9 | 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 that I have never worked on. |

Table 4 shows the needed skills to be able to complete successfully 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

In order 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 [29], [30]. Additionally, to enhance my decision-making abilities I have enrolled in a risk-specific course [31]. 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 passionate about. This activity will not only enhance my writing skills but also strengthen my critical thinking and ability to analyze and express complex ideas effectively.
* **Technical Skills:** Improving technical skills in our field typically involves two key approaches: studying documentation and hands-on 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 analyzing research that analyzes the performance of JSON and ProtoBufs, and the possible impacts of using ProtoBufs 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 Chapter 1.3 the research questions are:

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

RQ2. To what extent can ProtoBufs 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, that are reviewed by experts in the field.

## Search Terms

The keywords identified for the problem described are as follows:

* ProtoBufs
* Protocol Buffers
* 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

The *Code Snippet 4* was developed to systematically explore literature related to the use of Protocol Buffers and REST in the context of serialization and deserialization processes. The query incorporates key terms such as "JSON" and "XML" to compare serialization 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 optimization. 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 serialization.
* IC2: Research measuring energy consumption, CPU power usage, battery consumption, or memory usage associated with REST API communication.
* IC3: Studies providing serialization, deserialization, transmission efficiency, or resource usage metrics directly 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 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 [32]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 an important 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 [33]

## Discussion

Although Protocol Buffers are a well-established serialization technology, their application in RESTful architecture remains relatively uncommon. Most studies examining ProtoBufs focus on general 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, Protobufs could gain a broader adoption due to its high performance. As a result, we may witness a shift in serialization standards with binary serialization being embraced.

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

ProtoBufs are widely recognized for their performance and efficiency, offering advantages that make them particularly well-suited for high-performance applications. Unlike JSON, a text-based format, ProtoBufs employs a binary serialization 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 serialization formats, such as ProtoBufs, consistently outperform schema-less formats like JSON in terms of space efficiency. Even when JSON was compressed, ProtoBufs, also compressed, maintained its superiority in reducing data size, highlighting its adaptability to constrained environments, this phenomenon can be observed in the Table 8 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 ProtoBufs and Apache Thrift, outperformed JSON in serialization 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** | **ProtoBuf** | **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** | **ProtoBuf** | **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 ProtoBufs and Thrift significantly outperform XML and JSON.

ProtoBufs 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. ProtoBufs 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 ProtoBufs and Thrift.

The study made by Eduard Maltsev and Oleksandr Muliarevych[34] quantified ProtoBufs 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"[22]. This research evaluates Protocol Buffers within the context of RESTful microservices, specifically targeting Quarkus-based, cloud-native architectures. Not only that but also highlights ProtoBufs 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 ProtoBufs' capacity to optimize data interchange.

An observation from this research is ProtoBufs' 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 ProtoBufs 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 [22]

From Figure 8, it is evident that Protocol Buffers offer slightly better performance compared to JSON in the absence of caching within the same architectural context. 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 - Payload size benchmark with flat data

Reprinted from [22]

Figure 9 highlights the limitation of JSON serialization, in which, when subjected to scarce computer resources, JSON fails to serialize, resulting in the termination of its process. This failure underscores the limitations of JSON's text-based approach, 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 [22]

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 [22]

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 ProtoBufs 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 ProtoBufs generally surpasses 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.

ProtoBufs 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 ProtoBufs 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, ProtoBufs appears to be the superior choice. However, this efficiency comes with trade-offs, it was found that ProtoBufs, 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, ProtoBufs 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, ProtoBufs is shown to have increased energy consumption, which could outweigh its performance advantage over JSON. As a result, ProtoBufs are probably best suited for applications that prioritize performance and scalability, like in microservices or real-time systems with strict latency requirements. However, JSON may remain the preferred choice in scenarios that prioritize simplicity, flexibility, and ease of use.

In conclusion, while ProtoBufs 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, ProtoBufs can represent a promising alternative to JSON in HTTP-REST Based architectures.

# 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 serialization 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.

## Why Controlled Experiment Methodology?

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 serialization format (Protocol Buffers vs. JSON) on performance metrics such as speed and resource utilization [35], [36]. They ensure consistency and repeatability by applying the same conditions across tests, minimizing confounding factors. This methodology emphasizes internal validity, ensuring that observed differences are due to the serialization method rather than external influences like hardware or network variability [37]. By providing quantifiable metrics, it enables direct comparisons of key performance indicators and offers empirical validation to support theoretical claims about ProtoBufs efficiency.

## Data Collection

As a way of ensuring 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 Optimization.

# Analysis and Migration

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

## Project

The chosen project is a convenience store HTTP REST API developed in Java with the framework Spring Boot, which can be found on GitHub[38], 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.

### Context

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 criteria outlined in Section 6.1.

### Changes Made to Base Project

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

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.

# Implementation

The present chapter is concerned with the implementation details of the solution using Protobufs, 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 Protobufs to work properly, one of the first dependencies added to the pom.xml file is the *Protobuf-java* dependency, which allows us to use and process Protobufs 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 Protobuf 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.protobuf.ProtobufHttpMessageConverter;

@SpringBootApplication

public class *ConvenienceStoreApplication* {

    public static void *main*(String[] *args*) {

        SpringApplication.*run*(ConvenienceStoreApplication.class, args);

    }

    @Bean

    ProtobufHttpMessageConverter *protobufHttpMessageConverter*(){

        return new *ProtobufHttpMessageConverter*();

    }

}

Code Snippet 5 - Changes to allow Protobufs in project

After this, a plugin was added that would compile the Protobuf 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 Protobuf 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>protobuf-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.protobuf:protoc:4.29.3:exe:${os.detected.classifier}</protocArtifact>

</configuration>

</plugin>

</plugins>

</build>

Code Snippet 6 - Dependencies to compile Protobufs automatically

As illustrated in Code Snippet 6, the employed plugins were the os-maven-plugin and the protobuf-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 Protobuf files. Following the specification of the desired execution lifecycle, in this case the compile lifecycle, the plugin will then compile those proto files into Protobuf 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.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

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 Protobuf 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 Protobufs, 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 Protobuf 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/protobuf/java/com/conveniencestore/conveniencestore/Protobuf.

@PostMapping(produces = "application/x-protobuf", consumes = "application/x-protobuf")

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 protobufs

As demonstrated in Code Snippet 9, the necessary modifications to use Protobufs 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-protobuf”. Furthermore, to transmit a request body with a Protobuf, 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 Protobufs, 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 Protobufs

Lastly, in the *User* domain, introduce the new Protobuf classes into the service level, it's also straightforward, the Protobuf 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 to Protobufs. As illustrated in Code Snippet 12, the original version of the code segment under consideration exhibits a consistent structure, maintaining the preservation of the original steps. 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 protobufs

public UserResponseDTO insert(UserDTO data) {

if (userRepository.findUserByEmail(data.email()).isPresent()) throw new UserAlreadyExistsException();

String password = new BCryptPasswordEncoder().encode(data.password());

data = new UserDTO(data.username(), data.email(), password, data.role());

User user = new User(data);

user = this.userRepository.save(user);

return new UserResponseDTO(user.getId(), user.getUsername(), user.getEmail(), user.getRole(), user.getCreatedAt(), user.getUpdatedAt());

}

Code Snippet 12 - Portion of the control project to insert a new user at the service level

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 23 to 35. Subsequently, the Java classes are generated by the Protobuf compiler.

@GetMapping(produces = "application/x-protobuf")

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 13 - Example of one endpoint for *Product Entity* domain

The same principle is applied here, as in the User domain. As illustrated in Code Snippet 13, we have the changes to use Protobufs, in this case, a GET request is made for a list of *Product Entities*. Protobufs do not support Java lists as a native type. Consequently, a specialized proto is required for the list. As illustrated in Code Snippet 23, *ProductsEntityCatalog* is essentially a list of *ProductEntities*. The creation of *ProductsEntityCatalog* is done by simply calling the builder method and adding all the *Product Entities* with the *addAllProducts* method.

public List<ProductEntityOuterClass.ProductEntity> getAll(String orderby, String order) {

Sort.Direction direction;

switch (order) {

case "asc" -> {

direction = Sort.Direction.ASC;

}

case "desc" -> {

direction = Sort.Direction.DESC;

}

default -> {

direction = Sort.DEFAULT\_DIRECTION;

}

}

return productEntityRepository.findAll(Sort.by(direction, orderby)).stream().map(this::convertFromProductEntity).toList();

}

Code Snippet 14 - Example of service level method to retrieve all *Product Entities* with protobufs

As illustrated in Code Snippet 14, an example of a method migrated to Protobufs 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 JAR, and then converted it into a Docker image. These images were then disseminated to their respective Docker repositories [39], [40]. For reference, Code Snippet 15 contains the Docker file for both projects, Maven makes this process easier by generating a Java archive (JAR) file, which is an executable file that can be run in any environment.

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 15 - 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 [41], 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 [42]. 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 16, 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 17.

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 16 - 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 17 - Deployment manifest file for API

# Experiment

This section is going to provide insight into the experiments developed and the methodology used to evaluate the comparison between JSON and Protobufs. Furthermore, the goal question metric (GQM) approach[43] 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, this is important since there is a need to minimize confounding variables

## 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[43] 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 [43]

The central objective of this dissertation is to evaluate the performance of Protobufs 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 Protobufs versus JSON in HTTP based REST architectures to evaluate their performance and energy consumption”.

### Performance

Table 8 - Question and metrics for performance

| **Question** | **Metrics** |
| --- | --- |
| What is the difference in the performance of Protobufs 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 [44], 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 9 - Question and metrics for energy consumption

| **Question** | **Metrics** |
| --- | --- |
| What is the difference in energy consumption of Protobufs 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, and each directory inside the k6 directory contains the 3 repeated trials performed. It should also be noted that all graphs were created manually using Python. These can be found in the *Data\_Analysis* directory, which contains two Python files for constants, one for a normal Python file and the other for a Jupyter Notebook file. For better visualization, please refer to the *data\_csv.ipynb* file. Finally, to check the hypothesis testing, please refer to the *Hypothesis\_test.ipynb* file, which contains the statistical analysis with the distributions and the corresponding tests.

### Environment

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 10 - 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 |

### 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 one of the requests. It is worth noting that the tests were repeated 3 times for each endpoint [45]. 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 GET endpoint it would be sent 30,000 requests and, for each POST endpoint, it would be sent 10,000 requests, Table 11 shows all the tested endpoints for both JSON and Protobuf serialization methods. All the reports from the tests are available in the k6 directory, it contains not only the reports but also the scripts used.

Table 11 - Endpoints requested in the tests

| **Title** | **Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Create user | POST | /users | Performs a POST request to create 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 |

### Performance Tests

The following results are interesting, basically, they show that Protobufs on average have a higher response time 50% of the time, which is a bit questionable since it has a higher throughput 75% of the time. This phenomenon can be explained by the higher number of outliers in Protobufs. For instance, if we look for the median metric, it shows that, as the literature suggests, Protobufs are faster than JSON. A detailed examination of the data from the reports, reveals that the initial trial run for each serialization format consistently exhibits slower performance across all metrics. One potential explanation for this phenomenon is the Java Virtual Machine (JVM) [46], which has been observed to experience challenges during initialization, thereby affecting the efficiency of subsequent requests.

Table 12 - Benchmark results for creating Users

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 200.56 | 4.80 | 198.97 | 546.08 | 141.94 |
| Protobuf | 214.69 | 4.45 | 211.16 | 1116.44 | 152.56 |

Table 13 - Benchmark results for retrieving all Users

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 4.23 | 233.90 | 3.68 | 88.12 | 1.81 |
| Protobuf | 4.40 | 259.17 | 2.25 | 103.81 | 1.55 |

Table 14 - Benchmark results for retrieving User by id

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 3.44 | 371.55 | 2.34 | 392.56 | 0.96 |
| Protobuf | 3.36 | 383.12 | 2.27 | 650.04 | 1.01 |

Table 15 - Benchmark results for retrieving all Product Entities

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Average Response Time (ms) | Throughput (req/s) | Median(ms) | Maximum (ms) | Minimum (ms) |
| JSON | 3.65 | 286.79 | 3.07 | 194.59 | 1.50 |
| Protobuf | 3.28 | 313.59 | 2.75 | 92.64 | 1.37 |

Tables 12 to 15 present the aggregate metrics collected, encompassing all three trials for each benchmark. The results indicate that Protobufs demonstrate superior performance to JSON, albeit by a marginal margin. In the context of the system under consideration, Protobufs exhibit a slight advantage in all GET requests, with the exception of the POST request intended to create a user. In this particular instance, JSON exhibits a 7% improvement in average response time and throughput. To further analyze the results, the GET request to retrieve all product entities was examined. In this case, Protobufs demonstrated a 10% speed enhancement in the average response time and median, while the throughput showed an improvement of being 9% faster than JSON. Furthermore, the request to retrieve all users demonstrated that Protobufs have 4 % increased response time, but in the throughput, they show to be slightly faster than JSON by 11%, if we go further, it is possible to see that the median indicates that Protobufs have 12% lower response time, this required further analysis, and the conclusion made was that there are more outliers in the Protobuf samples, skewing the average response time, and for those reasons, the median might come as the more precise metric. Finally, the request to retrieve a User by its id, showed that the average response time of Protobufs is 2% lower than JSON, the throughput is 3% higher than JSON and the median response time is 3 % lower than the JSON.

This analysis demonstrated that Protobufs are indeed faster, but not in all cases. This phenomenon can be explained by the process being executed. For example, in the creation of a User, the system employs a hashing algorithm for the password field, which consumes resources and negates the benefits of supposedly having a faster serialization format. Furthermore, the analysis revealed that Protobufs exhibited marginal advantages in terms of faster execution times and reduced response times in select scenarios. However, in the context of retrieving all users, the average response time demonstrated that JSON outperformed Protobufs. This outcome was subsequently refuted through the examination of additional metrics that indicated Protobufs exhibited slight superiority over JSON. This discrepancy can be attributed to the presence of numerous outliers, which potentially skewed the average response time.

#### 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 [47] library, which offers all the algorithms for the statistical analysis. In essence, these tests serve to verify if the performance differences between JSON and Protobuf 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 10,000 requests, which means that a low number requirement test wasn’t possible, like Shapiro-wilk[48], as such, a test that doesn’t view the small deviations is important, the one chosen was *D’Agostinho and Pearson’s* [49], for two main reasons, the first one being that, the SciPy library uses that one as the default for the normality test [50], 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*[51], 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* [52], which works well with large datasets. Table 16 shows the created hypothesis for the performance study.

Table 16 - Hypothesis tests for each endpoint in terms of performance

| **Title** | **Method** | **Hypothesis** | |
| --- | --- | --- | --- |
| Create user | POST | Protobufs have a lower response time | Protobufs have higher throughput |
| Get all product entities | GET | Protobufs have a lower response time | Protobufs have higher throughput |
| Get all users | GET | Protobufs have a lower response time | Protobufs have higher throughput |
| Get user by ID | GET | Protobufs have a lower response time | Protobufs have higher throughput |

For all endpoints, the three repeated trials for each serialization format were aggregated. Code snippet 5 shows the base structure to perform these tests. The code and the results can all be found at */Data\_Analysis/Hypothesis\_test.ipynb*. It should be noted that the following code snippet involves looping through the aggregated data, which results in certain variables not being present within the shown code snippet.

## Test for normality

json\_normality\_p = normaltest(json\_response\_times).pvalue

protobuf\_normality\_p = normaltest(protobuf\_response\_times).pvalue

## Test the variance

variance\_p = levene(json\_response\_times, protobuf\_response\_times).pvalue

if json\_normality\_p > 0.05 and protobuf\_normality\_p > 0.05:

test\_stat, p\_value = ttest\_ind(

json\_response\_times, protobuf\_response\_times, equal\_var=(variance\_p >= 0.05)

)

test\_name = "t-test"

else:

test\_stat, p\_value = mannwhitneyu(

json\_response\_times, protobuf\_response\_times, alternative="two-sided"

)

print(f"Endpoint: {endpoint} ({test\_name})")

print(f"p-value: {p\_value}")

if p\_value < 0.05:

print("Significant difference found!\n")

else:

print("No significant difference.\n")

Code Snippet 18 - Response time hypothesis test structure

The results of the hypothesis tests for the response time hypothesis are interesting. There is no statistical significance for creating a User, which is consistent with prior analyses demonstrating the superiority of JSON in all metrics. The other requests, which are for the retrieval of data exhibited statistical significance, thereby substantiating the hypothesis that Protobufs demonstrate superior performance. In other terms, the null hypothesis (H0) was accepted for the creation of a User. However, the alternative hypothesis (H1) was supported by the results of the retrieval endpoint tests, indicating significant differences in performance.

Finally, it is important to note a few details about the throughput. The results align with the response times. For the creation of a user, the null hypothesis was accepted, indicating that Protobufs is not faster than JSON in this regard. However, the statistical tests indicated that the alternative hypothesis was supported for the remaining three hypotheses, indicating that, in the retrieval of a User by its ID, of Users and Product entities, Protobufs demonstrates superior performance in comparison to JSON.

### Energy Consumption Tests Setup

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[53], 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.(ADD LATER THE README)

### Energy Consumption Tests

The findings of the conducted experiments are quite intriguing, essentially only 25% of the cases show that Protobufs consumed more energy than JSON. However, in one instance where Protobufs demonstrated a total reduced energy consumption, specifically during the retrieval of all Product Entities, the average Joule consumption was marginally higher than that of JSON.

Table 17 - Energy consumption of User creation in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Total Joules | Total Average Joules | Average joules | Median Joules | Joules per Request |
| JSON | 30585.45 | 10195.15 | 1.53 | 0.02 | 1.02 |
| Protobuf | 18919.51 | 6306.50 | 1.46 | 0.02 | 0.63 |

Table 18 - Energy consumption of retrieval of all Users in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Total Joules | Total Average Joules | Average joules | Median Joules | Joules per Request |
| JSON | 2446.82 | 815.61 | 1.46 | 0.02 | 0.03 |
| Protobuf | 2206.28 | 735.43 | 1.26 | 0.02 | 0.02 |

Table 19 - Energy consumption of retrieval of User by its id in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Total Joules | Total Average Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1348.92 | 449.64 | 1.00 | 0.02 | 0.01 |
| Protobuf | 1400.43 | 466.81 | 1.06 | 0.02 | 0.02 |

Table 20 - Energy consumption of retrieval of all Product Entities in joules

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serialization Format | Total Joules | Total Average Joules | Average joules | Median Joules | Joules per Request |
| JSON | 1398.76 | 466.25 | 0.97 | 0.02 | 0.02 |
| Protobuf | 1334.70 | 444.90 | 1.03 | 0.02 | 0.01 |

Tables 17 to 20 present the aggregate metrics collected, as previously outlined in Section 8.2.3. Each table consolidates the metrics from the three trials for each specified endpoint. The findings suggest that Protobufs exhibit a reduced energy consumption compared to JSON, though the difference is marginal. The energy consumption of Protobufs appeared to decrease in proportion to the intensity of the serialization requirements. In this instance, the retrieval of a user by its ID, representing the most simple case of serialization, showed that JSON consumes slightly less energy than Protobufs, this is reflected in all the metrics observed, more precisely in the total joules metric, the total average joules, which is an average of the total joules consumed by each trial, the average energy consumption and the joules per request. In the context of the evaluation, it was observed that the median metric for Protobufs and JSON was equivalent, with a constant rounding to 0.02 joules. Additionally, the test case for user creation revealed that Protobufs consumed 38% less energy than JSON, with an average of 4% less energy consumption. This outcome is noteworthy given the observed challenges in performance during the load tests. In the retrieval of all users, Protobufs exhibited a 9% reduction in energy consumption compared to JSON, and the average joule metric also demonstrated a 13% decrease in energy expenditure. However, the retrieval of a User by its id, showed the opposite, it showed that Protobufs consumed a total of 3% more than JSON, and for the average joules consumed, the difference is about 6% increased energy consumption, this could be explained by the simplicity of the requests themselves, the only serialization process occurs in the return of the given User, which is a single entity with not that much data, and therefore the process of serializing it to Protobufs could incur in additional overhead, making it less energy efficient for small scale serializations. Finally, in the context of retrieving all Product Entities, it was observed that, in terms of total joules, Protobufs exhibited a 4% reduction in energy consumption compared to JSON. However, for the average joules, JSON demonstrated superior energy efficiency, with an average reduction of 6% in energy expenditure compared to Protobufs. This finding might be considered surprising, but when the duration of each test is considered, this suggests that, in this instance, Protobufs exhibited slightly superior speed, which can be validated in the previous performance tests, resulting in a net energy expenditure that was less than that of JSON, despite Protobufs's higher average consumption. It is noteworthy that JSON demonstrated greater efficiency in serialization compared to Protobufs in this case, as previously discussed.

#### 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, the *D’Agostinho and Pearson’s* algorithm 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 21 - Hypothesis tests for each endpoint in terms of energy consumption

| **Title** | **Method** | **Hypothesis** |
| --- | --- | --- |
| Create user | POST | Protobufs is more energy efficient than JSON |
| Get all product entities | GET | Protobufs is more energy efficient than JSON |
| Get all users | GET | Protobufs is more energy efficient than JSON |
| Get user by ID | GET | Protobufs is more energy efficient than JSON |

As illustrated in Table 21, 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 very interesting. Firstly, the creation of a user showed that there is significant statistical evidence that Protobufs are more energy efficient than JSON, which is confirmed by the data analyzed earlier. Additionally, the retrieval of all users further substantiated that Protobufs exhibit a substantial discrepancy in energy consumption compared to JSON. These findings lend support to the alternative hypothesis (H1). One potential explanation for this discrepancy could be the high intensity of serialization required. For instance, the retrieval of all users in JSON resulted in size of data received metric of 1926.08 MegaBytes (MB), whereas the Protobufs report showed 643.94 MegaBytes, which is also substantial. For the retrieval of a user by its Id and also for the retrieval of all product entities, the null hypothesis (H0) was accepted, which means that Protobufs are not as energy efficient as JSON in these cases.

## Conclusion

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

Initially, with regard to performance, Protobufs exhibited marginal superiority over JSON in three out of four of the evaluated endpoints. The sole endpoint where Protobufs 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, Protobufs demonstrated a clear advantage in terms of performance over JSON. However, it should be noted that the average execution time for Protobufs 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 Protobufs exhibit superior performance compared to JSON, if the right metrics are analysed.

Secondly, with respect to energy efficiency, Protobufs 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 Protobufs 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, Protobufs exhibit enhanced performance and demonstrate a potential correlation between performance and energy consumption. The analysis revealed that the faster Protobufs were, the less energy was consumed. This phenomenon was observed during the retrieval of all users and product entities. While Protobufs exhibited faster retrieval speeds, they also consumed more energy on average. However, when considering the total energy expenditure, Protobufs were found to be more energy efficient.

In conclusion, Protobufs 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 Protobufs exhibited 50% energy efficiency superiority over JSON in terms of average energy expenditure. However, when the total energy consumption was assessed, Protobufs 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, Protobufs has been observed to consume more energy than JSON. For a more detailed view of the processes that were run, refer to the repository [54] 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 Protobufs 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 Protobufs 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 Protobufs 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 Protobufs 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 Protobufs 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 Protobuf NPM package, which was essential for the execution of tests with Protobuf.

Another challenge that was encountered was during the integration process of the Protobufs, 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. However, through extensive research, despite the lack of available resources, the challenge was successfully addressed.

Despite the challenges posed by these challenges, their successful completion was both fulfilling and educational, giving me the necessary skills to effectively navigate future challenges.

## 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. Nevertheless, considerable effort was made to ensure that decisions were not influenced by personal preferences.

It is important to note the lack of, prior experience with Protobufs 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 Protobufs 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] “Open Science - European Commission.” Accessed: Dec. 02, 2024. [Online]. Available: https://research-and-innovation.ec.europa.eu/strategy/strategy-research-and-innovation/our-digital-future/open-science\_en

[15] 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

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

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

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

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

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

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

[22] 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

[23] 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.

[24] 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

[25] 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.

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

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

[28] 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.

[29] “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

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

[31] 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

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

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

[39] “1230199/control\_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/control\_project/general

[40] “1230199/experimental\_project general | Docker Hub.” Accessed: Mar. 13, 2025. [Online]. Available: https://hub.docker.com/repository/docker/1230199/experimental\_project/general

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

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

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

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

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

[46] B. Korando, “A Deep Dive into JVM Start-up – Inside.java.” Accessed: Mar. 19, 2025. [Online]. Available: https://inside.java/2025/01/28/jvm-start-up/

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

[48] “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

[49] “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

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

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

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

[53] “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

[54] “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

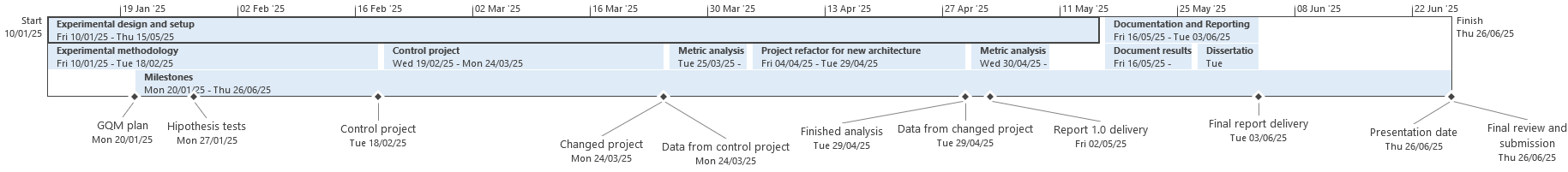


Figure 16 - Full timeline

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 22 - “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.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message UserDTO {

string username = 1;

string email = 2;

string password = 3;

UserRoles role = 4;

}

Code Snippet 19 - UserDTO generated proto file

syntax = "proto3";

package com.conveniencestore.conveniencestore.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message EditUserDTO {

string username = 1;

string email = 2;

}

Code Snippet 20 - EditUserDTO generated proto file

syntax = "proto3";

package com.conveniencestore.conveniencestore.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message LocalDateTimePb {

int32 year = 1;

int32 month = 2;

int32 day = 3;

}

Code Snippet 21 - LocalDateTimePb generated proto file

syntax = "proto3";

package com.conveniencestore.conveniencestore.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

enum UserRoles {

ADMIN = 0;

EMPLOYEE = 1;

}

Code Snippet 22 - UserRoles generated proto files

As illustrated in Code Snippets 19 to 22, the generation of the proto files for the User domain was facilitated by the IntelliJ plugin. However, the plugin could not automatically generate the packages, nor 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.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message ProductEntity {

int32 id = 1;

string name = 2;

LocalDateTimePb created\_at = 3;

LocalDateTimePb updated\_at = 4;

repeated Product products = 5;

}

message ProductEntityCatalog {

repeated ProductEntity products = 1;

}

Code Snippet 23 - ProductEntity generated proto file

syntax = "proto3";

import "local\_date\_time\_pb.proto";

package com.conveniencestore.conveniencestore.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message Product {

int32 id = 1;

int32 entity\_id = 2;

bool sold = 3;

LocalDateTimePb created\_at = 4;

LocalDateTimePb updated\_at = 5;

}

Code Snippet 24 - Product generated proto file

syntax = "proto3";

package com.conveniencestore.conveniencestore.protobuf;

option java\_package = "com.conveniencestore.conveniencestore.protobuf";

message ProductEntityDTO {

string name = 1;

}

Code Snippet 25 - ProductEntityDTO generated proto file

As with the *User* domain, the *Product Entity* domain generated proto files are visible in Code Snippets 23 to 25. One salient aspect to note is the *Product* proto, which even though it does not directly belong to the *Product Entity* domain, must be created because the *Product Entity* has a many to one relationship with it.