Kristianstad
University
Sweden

Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Independent project (degree project), 15 credits, for the degree of
Bachelor of Science (180 credits) with a major in Computer Science
Spring Semester 2021
Faculty of Natural Sciences

Enhance Inter-service Communication in Supersonic K-Native REST-based Java Microservice Architectures

Vincenzo Buono, Petar Petrovic

Author
Vincenzo Buono
Petar Petrovic

Title
Enhance Inter-service Communication in Supersonic
K-Native REST-based Java Microservice Architectures

Supervisor
Fredrik Stridh

Examiner
Dawit Mengistu

Abstract
The accelerating progress in network speeds and computing power permitted the architectural design paradigm to shift from monolithic applications to microservices. The industry moved from single-core and multi-threads, code-heavy applications, running on giant machines 24/7 to smaller machines, multi-cores single threads where computing power and memory consumption are managed very critically. With the advent of this novel approach to designing systems, traditional multi-tier applications have been broken down into hundreds of microservices that can be easily moved around, start, and stop quickly. In this context, scaling assumed a new meaning, rather than scaling up by adding more resources or computing power, now systems are scaled dynamically by adding more microservices instances. This contribution proposes a theoretical study and a practical experiment to investigate, compare and outline the performance improvements aid by the implementation of Protocol Buffers, Google's language-neutral, binary-based representational data interchange format over traditional text-based serialization formats in a modern, Cloud-Native, REST-based Java Microservice architecture. Findings are presented showing promising results regarding the implementation of Protobuf, with a significant reduction in response time (25.1% faster in the best-case scenario) and smaller payload size (72.28% better in the best-case scenario) when compared to traditional textual serialization formats while literature revealed out-of-the-box mechanisms for message versioning with backward compatibility.

Keywords
Enhance Interservice Communication, Protocol Buffers, Data Interchange Formats, Quarkus, Kubernetes

Contents

Acknowledgments

It is with genuine gratitude and warm regard that I dedicate this work to my whole family and I express my immense appreciation to my mother and my grandmother for supporting me along this journey and believing in me. – V. Buono

I would like to express my special gratitude to Sweden, my university (HKR) as well as our professors who allowed me to finish my studies.

All thanks to my partner, my great friend, Vincenzo Buono for all the teamwork we had as well as all the contribution he did to make this thesis astonishing.

Lastly, I would also like to thank my parents and friends who helped me and supported me throughout my studies – P. Petrovic

1 Introduction

This study proposes a novel approach to enhance inter-services communication in modern, *cloud-native*, *container-first,* REST-based *Quarkus* [1] microservice architectures by improving current, industry-standard text-based representational data interchange format and serialization technique. The outlined solution aims to improve *Machine to Machine communication* through the use of Google's *platform-neutral*, *language-neutral,* binary-based representational data interchange format, formerly named *Protocol Buffers* [2]. Throughout the thesis, the keywords *Supersonic* and *Subatomic*, where applicable, may be utilized as a substitute or synonym in reference to the *Quarkus* framework, as intended by the *Quarkus* authors [3]. *Quarkus* is a *Kubernetes-Native* Java stack tailored for OpenJDK HotSpot and GraalVM [1]. The word *K-Native* will be adopted during this study and implied as a shorthand for *Kubernetes-Native* and intended as a specialization of the *Cloud-Native* concepts, where applications are designed and built for *Kubernetes* specifically rather than deployed directly on the cloud [4].

1.1 Problem and Motivation

In the past decade cloud computing [5] has seen substantial growth, with REST-based (REpresentational state transfer) microservices being one of the most adopted *service-oriented architectures* (SOA) [6]. With this novel approach of designing a system as a collection of loosely coupled services, typically hosted on different remote nodes, that are interconnected through the use of lightweight mechanism over the network [7], the inter-service communication aspect started to gain attention due to its importance in the implementation of microservices. Notwithstanding the HTTP protocol capabilities allow the transmission of both textual and binary payloads, the widespread usage of text serialization formats such as JSON or XML is predominant over their binary counterparts, with JSON being the most used serialization format. The broad adoption of JSON has to be attributed to a multitude of reasons, but most importantly, to the fact that historically the primary consumer of APIs was a browser [8], where JSON is natively supported and a first-class citizen in the Javascript client-side programming language [9]. However, microservices are often implemented in different programming languages [10], running in different runtimes and environments, or leverage different technology stacks. In addition, a service might not be able to provide a complete solution to a specific request, and therefore compiling information with other domains may be necessary [11]; in such scenarios, where the *chained microservice design pattern* is applied, a single access point for a specific resource is provided, but its response is composed by aggregating multiple services' response. In such enterprise scenarios, performance metrics such as payload size, serialization and deserialization speed, transient memory usage during encoding (allocated and utilized heap and metaspace memory size), thread utilization and request/response latency, as well as maintainability factors such as strongly typed and schema-based data format with improved supports for versioning are essential for the deployed system to respect the required *service-level agreement* (SLA) [12,13].

In this context, it is logical to be questioning whatever the communication between services can be improved by adopting a platform-neutral, binary-based representational data interchange format and an improved serialization approach in professional Java Cloud-native microservice architectures built on top of Quarkus [1] and GraalVM [14,15].

1.2 Research Questions

The study answers the following research questions:

R1: *From a performance standpoint, how can the response time and payload size be reduced in a modern, Supersonic, Cloud-Native, K-Native REST-based Java Microservice endpoint?*

R2: *How can a strongly typed binary-based representational data interchange format with schema support improve versioning management, data evolution, and migration in microservices?*

1.3 Aim and Purpose

The purpose of this research is to analyze, investigate and enhance inter-services communication in modern, *cloud-native*, *container-first,* REST-based *Quarkus* [1] microservice architectures by improving current, industry-standard text-based representational data interchange format and serialization technique.

The following thesis will discuss in great detail how to improve Machine to Machine communication through the use of a platform-neutral, binary-based representational data interchange format and an improved serialization approach in professional Java Cloud-native microservice architectures. Furthermore, findings will be proposed to demonstrate how the use of *Protocol Buffers* [2] can enhance such systems by providing a smaller payload, faster serialization and deserialization, better versioning, and support for typed data compared to traditional, schema-less data interchange formats such as JSON.

Protocol Buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data [2].

1.4 Limitations

1.4.1 K8s deployment

In order to study and investigate the impacts of the usage of a binary-based representational data interchange format, in modern containerized *Quarkus* microservices, the services have to be deployed on a Kubernetes [16] cluster. Deploying microservices on the cloud, however, especially for benchmarking purposes, can be very complex as well as very expensive due to the typical *pay-per-consume* rates, as the customer typically pays for the computational power that the services use. For this reason, and also because the research would exceed the scope and thesis constraints, the microservices have been deployed locally with the use of third-party software to ensure the validity of the gathered data, as well as the accuracy of the results in a real deployment scenario. In this regard, *KIND* (Kubernetes IN Docker) [17] has been adopted to ensure the portability of the results in the cloud, as described in section **4.1.**

1.4.2 Data interchange formats

For the following study has been chosen, as a reference binary-based data interchange format, Google's language-neutral, platform-neutral, *Protocol buffer* serialization format. As shown in section **2.3.2**, there are many binary serialization formats, even with faster serialization speeds than *Protobuf*, as illustrated in **Figure 7.** *Protocol Buffer* has been chosen among those due to its support and popularity, but most importantly, it is natively supported by *Quarkus* as a native extension. *Quarkus'* native extensions differ from regular libraries because they can be natively compiled and optimized by the GraalVM when running as a native executable. Since enhancing inter-service communication is a prerogative of this thesis, it has been recognized to be essential to pick technologies that do not decrease or degrade the efficiency or performance as it would go against the aim of this research.

1.5 Ethics

Since the results contained in this thesis can ultimately be utilized either as a consideration factor during the design of a REST microservice architecture or as a starting point for further work in the field, it has been determined that the experiments contained in this study are conducted ethically. In this context, the benchmarks provided along with this study have been created with the aim to investigate and measure the impact of the implementation of *Protocol Buffers,* Google's binary-based representational data interchange format, over traditional textual serialization formats in a modern, Supersonic, Cloud-Native, REST-based Java Microservice architecture. With the end goal of enhancing interservice communication, we specifically limit the scope of our analysis and testing to the technologies previously discussed, isolating the improvements aid by the latter from other possible contaminating factors. However, as broadly discussed in section **3.3** and subsequently confirmed by the findings provided in section **5.2**, the results obtained can vary based on the testing environment, on the version of the technologies analyzed as well as the complexity of the encoded data structures. For these reasons has been disclosed a public listing of all the technologies and versions used during the execution of the empirical experiment along with the hardware specification of the testing rig (see section **3.4**).

2 Background

In recent years, there has been an increasing interest in cloud computing, with REST-based microservice architectures being one of the most adopted *service-oriented architectures* (SOA) [18]. This novel approach of designing a system as a collection of loosely coupled *services* aims to break up traditional code-heavy monolithic multi-tier application architectures into independently self-contained, deployable services that can be seamlessly stopped, started, and scaled in the cloud accordingly to the load balancing demands [8]. Layered architectures, as depicted in **Figure 37** (see **Appendix 1: Microservices**), have each layer tied to a different service boundary, with each being related to a specific technical solution; in such scenario, implementation of new functionalities is expensive and fairly inefficient as changes are often not bounded to a specific technology stack and therefore span over multiple layers. In contrast, a *service* in a microservice architecture is modeled around a business domain and contains end-to-end slices of business functionalities [8] by encapsulating, where applicable and necessary, all the layers including, but not limited to, presentation layer, business layer, persistence layer, and data storage layer as illustrated in **Figure 38** (see **Appendix 1: Microservices**). Systems built upon the *microservice architectural style* are designed as a suite of small interconnected services, each running on its own process, typically within an orchestration framework, that encapsulates a business functionality that is made accessible to other services via network [8] through the use of lightweight mechanism such as, but not limited to, HTTP resource API [7]. Microservices provide their business functionalities on one or more network endpoints [8] and hand over their resources serialized into a specific data interchange format that can be deserialized and consumed by the clients. Typical REST microservices use a text-based representational data interchange format and serialization technique due to the high compatibility of the latter in the destination platform that the technology originated and evolved: the web. Current industry-standard formats are JSON and XML [19].

2.1 Microservices

The microservice architecture has become one of the most adopted and predominant architectural style in the *service-oriented* industry [18]. The microservice architecture is an architectural style designed as a collection of small services that suites large-scaled applications. Attributes such as loose coupling, high cohesion, resilience, and scalability are typical of microservices architectures.

Services are built upon business functionalities and are platform agnostic, meaning they are commonly built using numerous programming languages. Microservices often communicate through HTTP mechanisms such as REST (Representational State Transfer). Since each service is independent of the other, they do not need to know about the underlying architecture or the implementation [20,21].

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. [...] Services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages." [7]

2.2 Quarkus

2.2.1 Supersonic, Subatomic, Java

Quarkus [1] is an open-source *cloud-native* framework with built-in Kubernetes [22] integration [23]. The open-source stack is often referred to as "supersonic, subatomic" as *Quarkus* tailors your applications to minimize boot time with particular attention to the *First Response time,* allowing requests to be serviced in the order of milliseconds [24], as shown in **Figure** 1. The framework is also optimized to have a small memory footprint, allowing low main memory or *resident set size* (RSS) consumptions while producing small binaries, hence the word "subatomic" in the framework definition, as illustrated in **Figure 2**. *Quarkus* is also polyglot because it provides supports to several JVM languages and, thanks to *GraalVM,* allows the developers to write *polyglot applications* that seamlessly can pass values from one language to another [25] by means of the *Truffle language implementation framework* [26].

A Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards.[1]

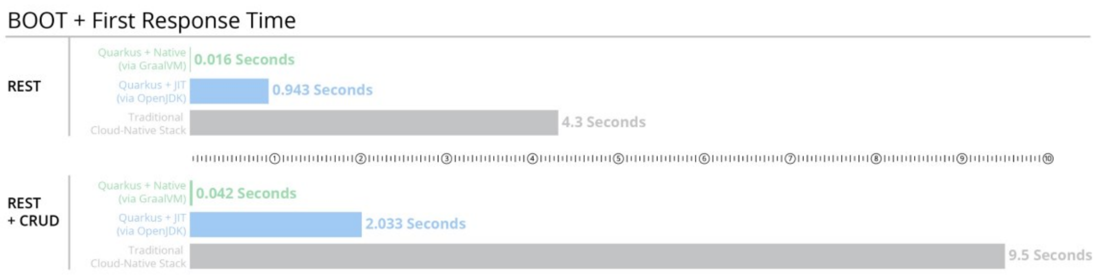


Figure 1 - Quarkus Boot + First Response Time official benchmarks [1]

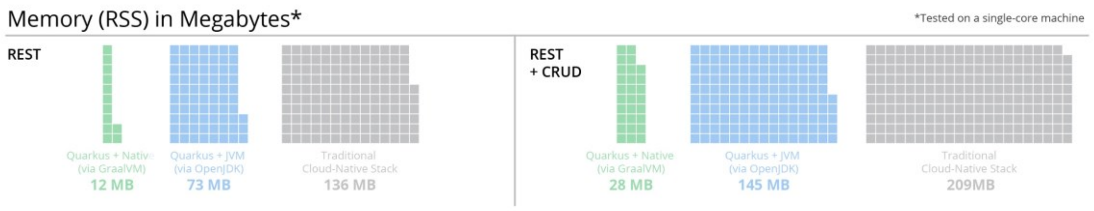


Figure 2 - QuarkusMemory (RSS) in Megabytes official benchmarks [1]

2.2.2 Container-First

Quarkus has been designed around a container-first philosophy [27], allowing developers to create, optimize and deploy cloud-native applications in Kubernetes' container clusters with ease by providing all the previously described advantages.

When deploying a *containerized microservices* [10], the end goal is to maximize the number of instances of your application in order to scale and meet the unexpected load while still utilizing as many of the resources as possible. During the scaling-up process, in order to fulfill the incoming traffic load, the application's instances need to be up and running as quickly as possible; Quarkus, by producing small, native executable binaries, through the use of *GraalVM,* solves this problem, creating container-native applications for peak performance [23,24]. Its performance is attributed not solely to the *First Class*

*Support for Graal/SubstrateV*M [27], but also to the optimized *Build Time Metadata Processing* [28] executing as much processing as possible at build-time rather than at runtime resulting in less memory usage and faster start-times as only the classes that are needed are loaded into the production JVM. *Quarkus* also optimizes Reflection [29] by reducing its usage or, in some instances, even avoiding it altogether. In addition, using a technique called *Native Image Pre Boot* [30,31], *Quarkus* pre-boot as much of the framework as possible during the native image build process, allowing the resulting native image to have executed most of the startup code, that otherwise a typical application would need to run at every application's spin-up, and serialized the result into the final executable, resulting in faster startup times [27]. **Figure 3** illustrates the native image build process previously described. **Figure 4** illustrates a simplified representation of GraalVM architecture in relation to the *Truffle Framework*.

Providing native support to Kubernetes, thanks to its built-in Docker images and Kubernetes extensions, *Quarkus* is defined as "K-Native" or "Cloud-Native".

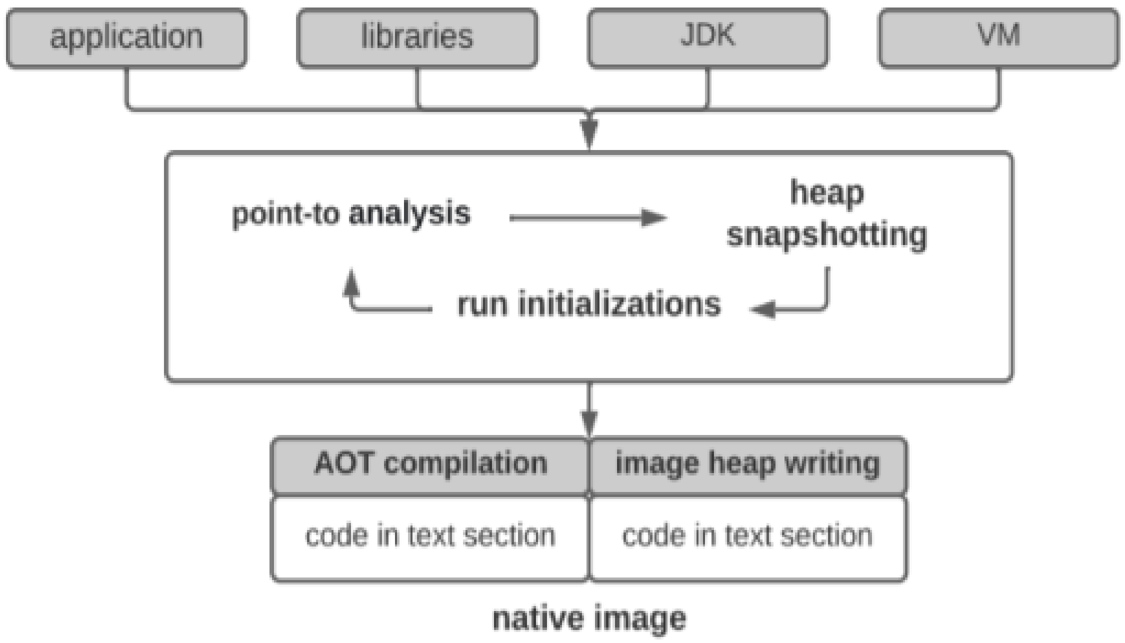


Figure 3 - Image build-time process [31]

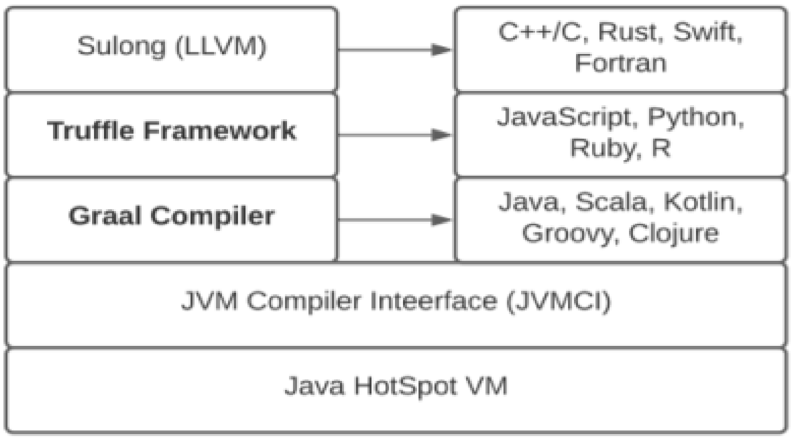

Sulong (LLVM)
C++/C, Rust, Swift, Fortran
Truffle Framework
JavaScript, Python, Ruby, R
Graal Compiler
Java, Scala, Kotlin, Groovy, Clojure
JVM Compiler Inteerface (JVMCI)
Java HotSpot VM

Figure 4 - Simplified representation of GraalVM Architecture and Truffle framework [31]

2.3 Data interchange formats

Efficient mechanisms for data structuring and formatting are indispensable for managing data traffic between services in order to ensure portability and interoperability while avoiding excessive bandwidth costs [32]. Serialization formats or data interchange formats directly impact microservices performances by (I) defining the end payload size of the HTTP request and HTTP response carried in the body section, (II) affecting the service's latency or response time as the server's serialized or deserialize the data [33]. Data interchange formats can be classified based on their typology: Textual formats and Binary Formats [8].

2.3.1 Textual formats

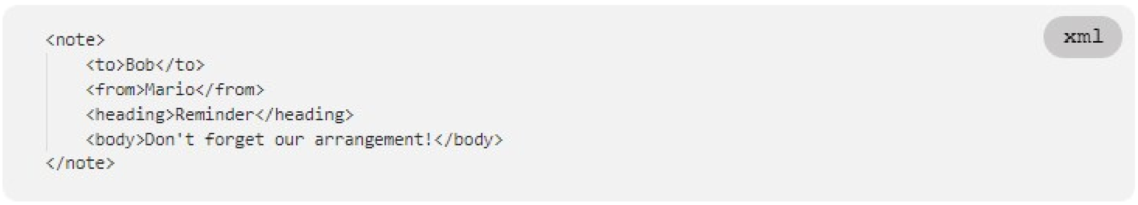
Text-based data interchange formats are defined as such due to their textual representation after the serialization process. The widespread adoption of textual serialization formats guarantees high interoperability and flexibility [34]. This class of serialization formats is characterized by the ability to be *human-readable* [8] and *human-writable* [32] at the expense of schema support and typed data. The most notable textual interchange formats are *XML* and *JSON* [35].

eXtensible Markup Language (XML) is a widely used standard format for data representation in applications, including Web Services, and is designed to provide

simplicity, generality, and usability of data exchanged over the Internet [36]. **Figure 5** illustrates the data structure of an uncompressed XML message.

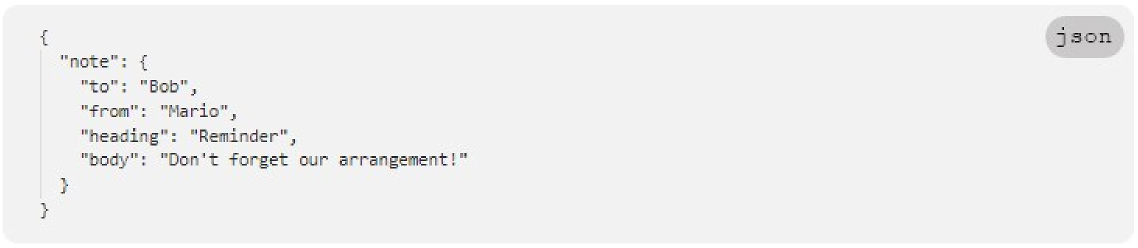
JSON [34] is a text-oriented, lightweight, *human-readable* data interchange format designed for the representation of simple data structures and associative arrays where messages can be organized as objects composed of key-value pairs or an array of objects [36]. **Figure 6** shows the data structuring of an uncompressed JSON message.

The past decade has seen an increase in JSON utilization as a textual data interchange format, replacing XML in many fields and becoming the industry-standard serialization format for the implementation of REST APIs [8]. It also has proven to provide better performance in terms of speed and less resource usage, even in a context where JSON is not native [33].



```xml
<note>
    <to>Bob</to>
    <from>Mario</from>
    <heading>Reminder</heading>
    <body>Don't forget our arrangement!</body>
</note>
```

Figure 5 - XML Data Structing format (uncompressed)



```json
{
  "note": {
    "to": "Bob",
    "from": "Mario",
    "heading": "Reminder",
    "body": "Don't forget our arrangement!"
  }
}
```

Figure 6 - JSON Data Structuring (uncompressed)

2.3.2 Binary Formats

Binary data interchange formats trade *human readability* and *interoperability* in favor of faster serialization times and smaller payload size, as the data structure, after being encoded, is represented in native binary form [8]. This class of serialization protocols varies greatly, but they all aim to provide built-in schema support and validation and strongly typed data support.

The past years have seen rapid development and interest in such technologies thanks to their speed and size [8]. Currently, the landscape is vast, with multiple different formats available such as Protocol Buffers, Cap'n Proto, FlatBuffers, and many more, as illustrated in **Figure 7** in conjunction with their serialization and deserialization performances. **Figure 8** shows the space allocated after serialization, with and without compression.

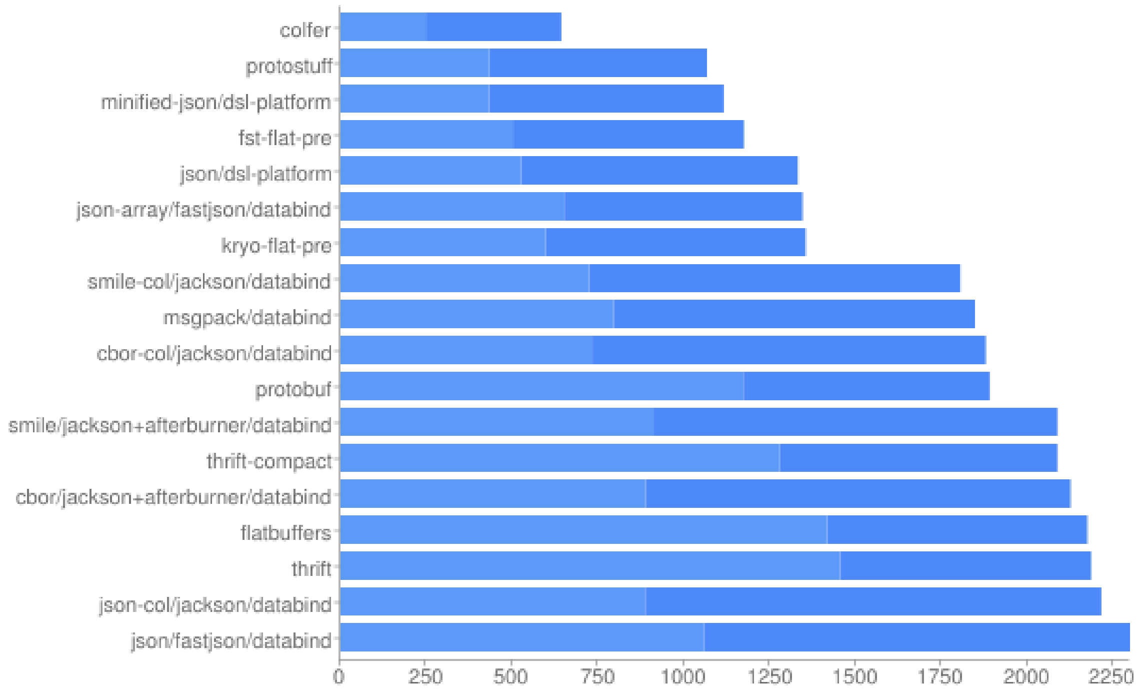


Figure 7 - Total Serialization + Deserialization Time of common Binary Serialization Formats [37]

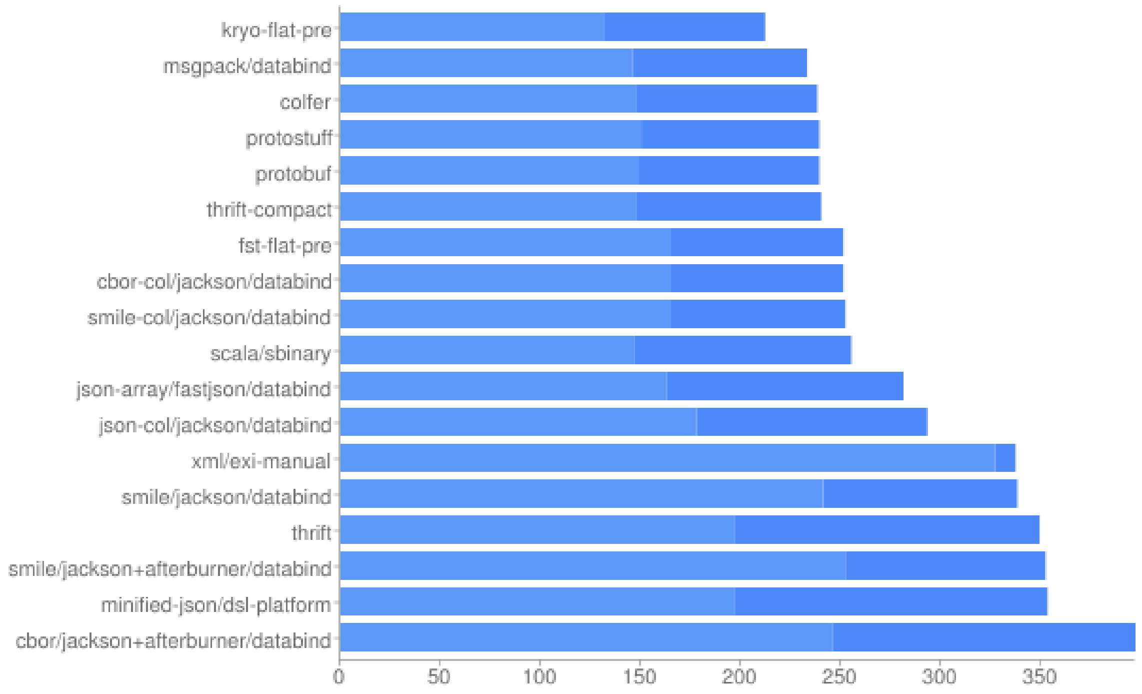


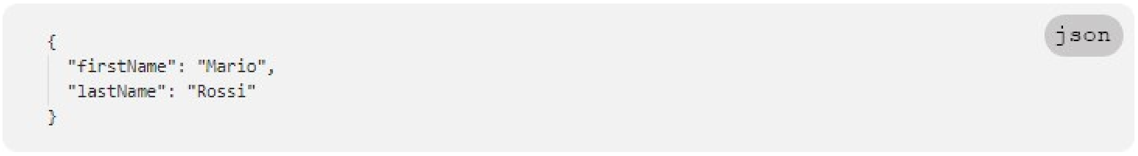
Figure 8 - Total allocated space of common Serialization Formats. Light blue shows the compressed size [37]

2.4 Protocol Buffers

Protocol Buffers, formerly known as *Protobuf*, are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data [2]. Protocol Buffers have been designed with speed and space efficiency in mind, providing fast serialization and deserialization times while retaining a small memory footprint for efficient network bandwidth usage. It comes with built-in mechanisms for versioning the APIs or endpoint interfaces, allowing the data contract to seamlessly evolve without breaking applications that are still built on top of legacy versions of the API. The structure of the data to be encoded is defined by the user through the definition of *.proto* file[36]. The user-defined data structure is subsequently compiled using the Protocol buffer compiler [38].

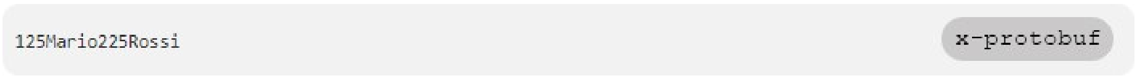
2.4.1 Context and Data Separation

Protobuf separate the context of the data from the data itself by defining the structure of the *message* in a separate config file with *.proto* extension [2], allowing to reduce the payload size as the transmitted *message* contains only the data and not the context. This is not only important from a performance standpoint but also allows to validate the *message* against an interface. It can be observed in **Figure 10** and **Figure 9** the different message lengths of the same message encoded in JSON and using the Protobuf compiler, as the JSON message has to transmit the context and the data, increasing the total message size.



```json
{
  "firstName": "Mario",
  "lastName": "Rossi"
}
```

Figure 9 - Example of a JSON encoded object

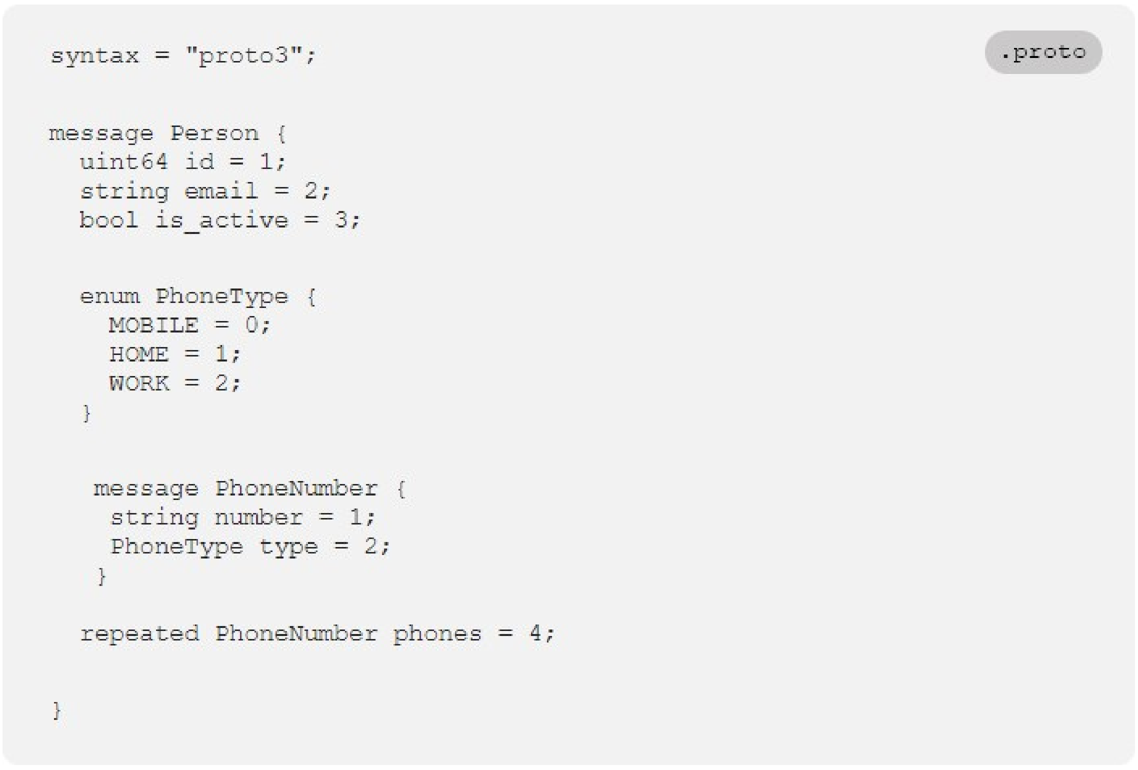


```
125Mario225Rossi
```

Figure 10 - Example of a Protobuf encoded object in textual representation

2.4.2 Message Format

The data is encoded using Protobuf based on a user-defined configuration known as *messages* [2]. Each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers, booleans, strings, raw bytes, or other protocol buffer message types [36]. **Figure 11** shows an example of a message type definition of a Person object.



```
syntax = "proto3";

message Person {
  uint64 id = 1;
  string email = 2;
  bool is_active = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

   message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
   }

  repeated PhoneNumber phones = 4;

}
```

Figure 11 - Example of a Message Type definition [39]

2.4.3 LEB128: Base 128 Varints

Base 128 Varints are a method of serializing integers using a variable length of bytes [40] and are utilized to overcome the problem of representing varying length integer values [41]. This technique replaces the need to use a delimiter byte by using a single bit and is part of a family of encoding techniques referred to as *variable-length encoding* [42]. The encoded integer has the most *significant bit* (MSB) of every octet set, besides that last byte. This approach allows to signal whatever there are more byte to be read. The remaining lower 7 bits of the octet are used to store the two's complement representation of the number in group s of 7 bits with the *least significant group first* [40]. **Figure 12** shows an example of the number 300 encoded with the following technique.

```
1010 1100 0000 0010                                        LEB128
```

Figure 12 - Example of the number 300 (base 10) encoded with LEB128 [40]

2.4.4 Binary Encoding

An example of a *Person* message, as shown in **Figure 13**, encoded using the *protoc* compiler has been illustrated in **Figure 14.**

```
message Person {                                                         .proto

    required string user_name = 1;

    optional int64 favorite_number = 2; repeated string interests = 3;

}
```

Figure 13 - Simplified Schema of a Person object

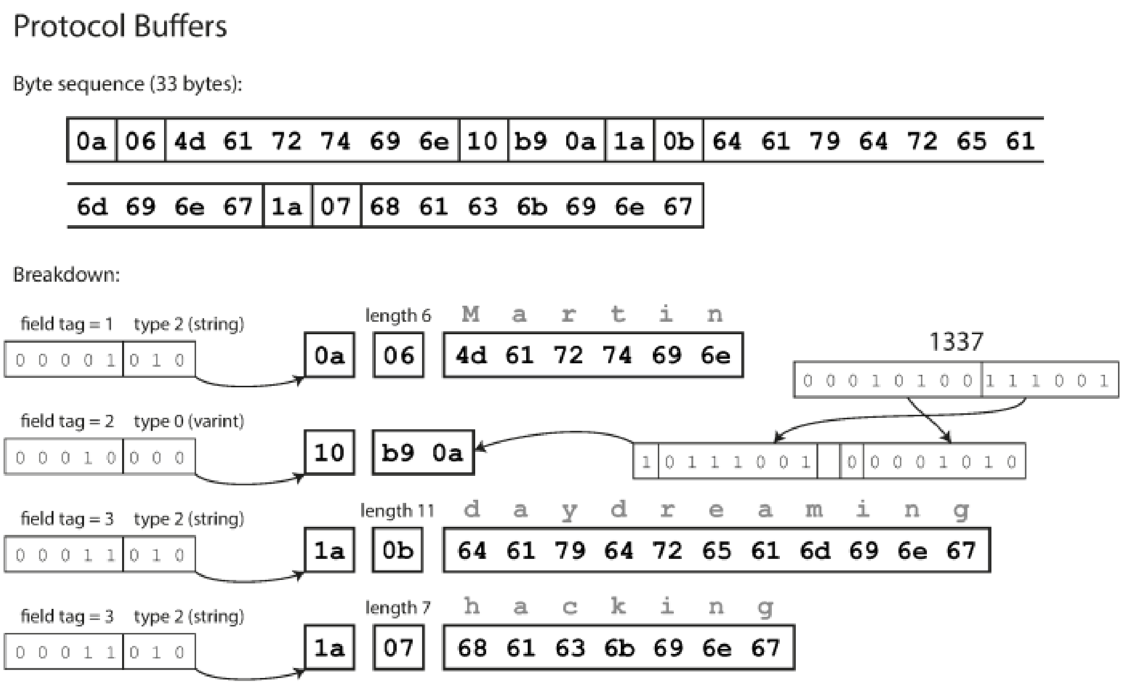


Figure 14 - Protobuf Encoded de-structuring of the Person object based on the Person Message Schema [43]

3 Methodology

This study has been conducted from a theoretical standpoint, through the use of literature review, as well as from a practical standpoint through the implementation of an experiment that aims to test, compare and outline the performance improvements aid by the implementation of *Protocol Buffers,* Google's language-neutral, binary-based representational data interchange formats over traditional text-based interchange format in a modern, Supersonic, Cloud-Native, K-Native REST-based Java Microservice endpoint. This section covers the details of the former, while the experiment is only briefly introduced; more information is provided and discussed in the **Benchmarking** chapter of this study.

3.1 Information Sources

To gather all the relevant information on the investigated subject area, as well as all the related work and the previous studies conducted on the topic, broad and detailed research has been conducted utilizing the following tools and platforms:

1. IEEE Xplore [44]
2. SciTePress [45]
3. Google Scholar [46]
4. ACM Digital Library [47]
5. Kristianstad University Library [48]

The above-stated resources have been sorted in descending order by degree of relevance of the articles used for the following study.

3.2 Literature Search Criteria

In order to gather pertinent results during the literature research phase of the study, a refined selection of keywords has been adopted. The first approach has been to utilized keywords directly related to the research question in order to yield and obtain the most accurate and relevant studies on the subject. Therefore, a combination of the keywords "data interchange format", "cloud-native", "k-native", "serialization formats", "inter-service communication" was used. However, as a result of this search, very few results have been obtained with no specific material on the implementation or usage of binary-based data interchange formats in REST microservices in the context of Kubernetes clusters. This motivated us to conduct a primary research with the help of an empirical experiment to gather relevant data on the subject, as explained more in detail in section **4.0 Benchmarking.**

3.3 Empirical approach

The approaches provided in this thesis, aimed at improving current inter-service communication in high-density, K-Native Java REST-based microservice architectures, will be implemented, analyzed, and compared to the current industry-standard technology. Therefore, multiple REST API endpoints [5] will be created and tested utilizing both text-based and binary-based representational data interchange format through the use of *Quarkus* and *GraalVM*; varying data structure complexity, payload size, and request frequency against the different approaches will allow to investigate the provided approaches performances. Moreover, the testing will outline and represent material for discussion regarding the benefits and caveats of each approach while highlighting the implication that the usage of binary-based representational data interchange format carries and how this affects the serialization and deserialization performance.

3.4 Testing environment

The data collected during the following study has been gathered in a controlled environment, and each test has been executed subsequently to one another. A complete list of all the software used, with associated versions, is shown in **Table 1**, while the hardware specifications are shown in **Table 2**.

Table 1 - Sofware specifications

No	Technology	Name	Version
1	Operating System	Windows 10 Pro	v10.0.19042 Build 19042
2	k8s cluster	Kind	v0.10.0
3	Container Orchestration	Kubernetes	v1.14.0
4	Java Stack	Quarkus	v1.13.3.Final
5	Binary serialization format	Protobuf	v1.8.1 (camel-quarkus-protobuf)
6	RESTful Framework	RESTEasy	v1.13.3.Final
7	JSON Processor	Jackson	V1.13.3.Final
8	Programming Language	Java	SDK v14.0.1
9	Java Runtime (VM)	GraalVM	GraalVM CE 21.1.0
10	JSON Compressor	Gzip	v2.3.2.Final
11	Container Platform	Docker	C.19.03.9 | S.19.03.8
12	Dummy Data generator	Faker	V0.15

Table 2 - Hardware specifications

No	Part	Model
1	CPU	Intel Core i9-9900K (multi-threading enabled)
2	Motherboard	MSI MPG Z390 GAMING EDGE AC
3	GPU	GeForce RTX 2080 @ 1860MHz
4	Ram	64GB @ 3600MHz
5	Storage	Samsung SSD 970 Evo Plus

3.5 Analyzed metrics

The practical implementation of the following study will investigate the performance impacts of the usage of a binary-based data interchange format, such as Google's *Protocol buffers* [2], in the inter-service communication between two Java REST microservices endpoints in a local Kubernetes cluster. Consequently, as explained in section 1.3 and in more detail in section 2.3, the microservice's response time and latency, in relation to the serialization times, in each and every respective studied serialization formats, will be investigated as well as the HTTP response payload size with and without gzip [49] compression. *Transient memory*, as well as RSS during serialization, will also be analyzed, as been recognized to majorly impact microservices' costs.

4 Benchmarking

This section aims to provide a more in-depth overview of the characteristics, components, and tools utilized during the execution of the practical experiment and the specifics regarding the data collection and its implications.

4.1 K8s Local Cluster Environment

To ensure the study's validity and the required level of data accuracy, as discussed in section **1.4.1**, the experiment has been conducted in a local *Kubernetes* (K8s) cluster to simulate a deployment of a *modern containerized microservice*. The K8s cluster has been implemented using *KIND* (Kubernetes IN Docker) [50] and the *kubernetes-cli (kubectl)* [51].

"kind is a tool for running local Kubernetes clusters using Docker container "nodes" [17]

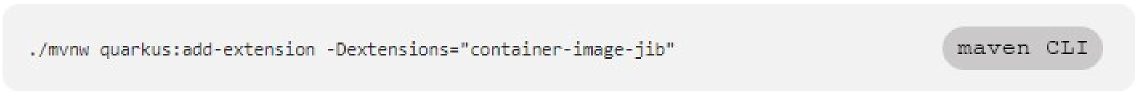
4.2 Quarkus Microservice Deployment

The *Quarkus* microservices have been deployed and run on the K8s cluster within a *Docker container* [52] in *JVM mode* [53] as well as in *native executable mode* [54]. Both the deployment have been investigated, but their performance difference has only been discussed in section **6.1**, whereas it intersects with the subject of this study or directly impacts one of the analyzed metrics, as discussed in the study limitations in section **1.4**.

4.2.1 JVM-Mode Container

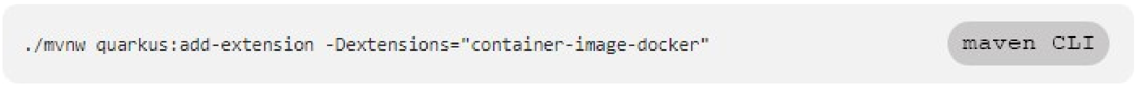
The *Quarkus* microservices containers, regarding the creation of a containerized version running in *JVM mode,* have been built using the built-in *Dockerfile.jvm* (see **Appendix 1: Docker**). The container image build has been performed using *Jib* [55] along with the *Quarkus* extension *quarkus-container-image-jib* (**Figure 15**). The extension *quarkus-container-image-docker* (**Figure 16**) has been used on top of Docker binary. **Figure 17** shows the maven commands used to set the *quarkus.container-image.build=true* flag in order to build the image.

For the full docker file, please refer to **Appendix 2: Docker build**.



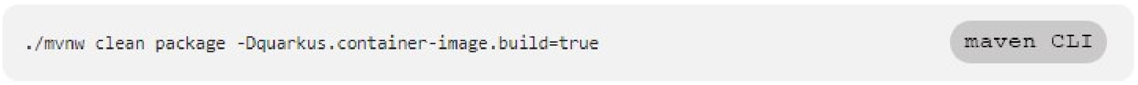
```
./mvnw quarkus:add-extension -Dextensions="container-image-jib"
```

Figure 15 - Maven: adding quarkus-container-extension [56]



```
./mvnw quarkus:add-extension -Dextensions="container-image-docker"
```

Figure 16 - Maven: adding quarkus-container-image-docker extension for Docker binary [56]



```
./mvnw clean package -Dquarkus.container-image.build=true
```

Figure 17- Maven: building container image [56]

4.2.2 Native-mode Container

The *Quarkus microservices* have also been compiled to a native executable [30] and then packaged within a container. The native executable has been built using *Mandrel* [57], a distribution of GraalVM CE. For the full docker file, please refer to **Appendix 2: Docker build**.

4.3 Supplementary Tools

A list of supplementary tools and software utilized to inspect, collect, or produced the data contained in this study has been supplied below.

1. **API Testing**: Postman [58]
2. **API load testing and graphing**: JMeter [59]
3. **Memory and performance analysis**: JConsole [60]
4. **Sampling, profiling, and tracing**: VisualVM [61]

4.4 Test design

The tests have been designed to investigate and enhance inter-services communications in modern, *cloud-native*, containerized RESTful *Quarkus* microservices by utilizing Google's binary-based data interchange format and compiler, as explained in section **1.3**. As part of the investigation, two containerized microservices have been created and made communication through the use of lightweight REST-based mechanisms. For each testing scenario (see section **4.7**), multiple subsequent requests have been executed in the magnitude shown in **Table 3** while varying the request type, the complexity of the data structure serialized, and the count of the generated data (see section **4.8**).

Table 3 – Testing conditions

Data structure design	Request count[1]	Request types[2]		Data generation count[3]
Flat (simple)	1, 5, 50, 100	GET	POST	Up to to 300 000
Nested (complex)	1, 5, 50, 100	GET	POST	Up to 650

Note[1]: Applied only to metrics which result is measured in units of time.
Note[2]: Indicates the HTTP method upon which the experiment is conducted.
Note[3]: The date generation count has been set to create edge-case scenarios as well as an average scenario.

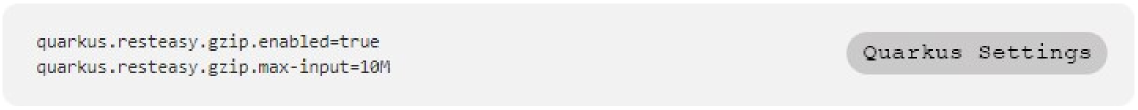
4.5 Serialization formats

The analyzed and compared serialization formats are *JSON,* as a representative format for the textual category, and *Protobuf* for the binary-based class.

The *JSON* data has been encoded using the *Jackson* [62] serializer (see section **3.4** for more details). On the other hand, the *Protocol buffer* data has been encoded using the official c++ compiler.

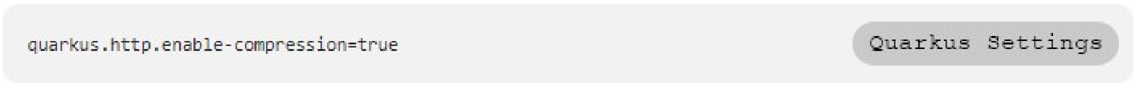
4.6 Compression

The serialized data has been tested with compression enabled and disabled, using Gzip (see section **3.4** for more info). The compression has been executed prior to the data being transmitted over the wire, and the payload has been compressed through the built-in *RESTEasy* Gzip and a more efficient external implementation. An upper limit on deflated request body has also been applied as *Quarkus* specification recommends [63]. **Figure 18** shows the Gzip being enabled in the *Quarkus* application config file, while **Figure 19** illustrates the HTTP compression being enabled globally.



```
quarkus.resteasy.gzip.enabled=true
quarkus.resteasy.gzip.max-input=10M
```

Figure 18 - Quarkus application settings: Gzip support enabled [63]



```
quarkus.http.enable-compression=true
```

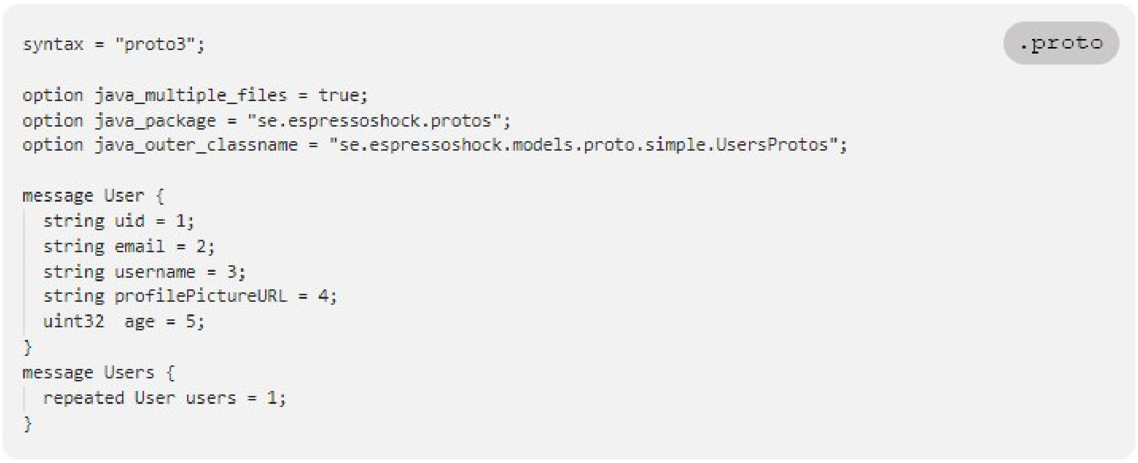
Figure 19 - Quarkus application settings: Global compression setting

4.7 Data Structure complexity

In order to investigate the metrics proposed in section **3.5,** the REST requests have been composed by varying the encoded data structure. Two data structures with different complexity have been designed and implemented in the benchmarks to evaluate and highlight the impact that different data structure complexity has on the *serialization speed*, and ultimately on the *response time*, as well as studying which method has better CPU utilization as well as lower transient memory usage.

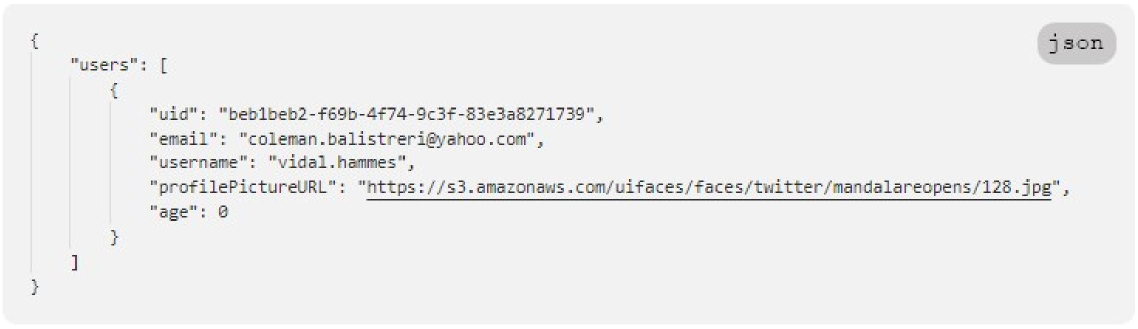
4.7.1 Flat Approach

The first tested data structure has a simpler composition, as it can be observed in Figure 21, with a flat approach and no nested objects. It consists of a simple *User* object, with the following fields: *uid*, *email*, *username*, *profilePictureURL*, and *age*. Figure 20 illustrates the proto definition used, while Figure 21 depicts an extract of the serialized JSON object collection.



```
.proto
syntax = "proto3";

option java_multiple_files = true;
option java_package = "se.espressoshock.protos";
option java_outer_classname = "se.espressoshock.models.proto.simple.UsersProtos";

message User {
  string uid = 1;
  string email = 2;
  string username = 3;
  string profilePictureURL = 4;
  uint32  age = 5;
}
message Users {
  repeated User users = 1;
}
```

Figure 20 - User proto definition

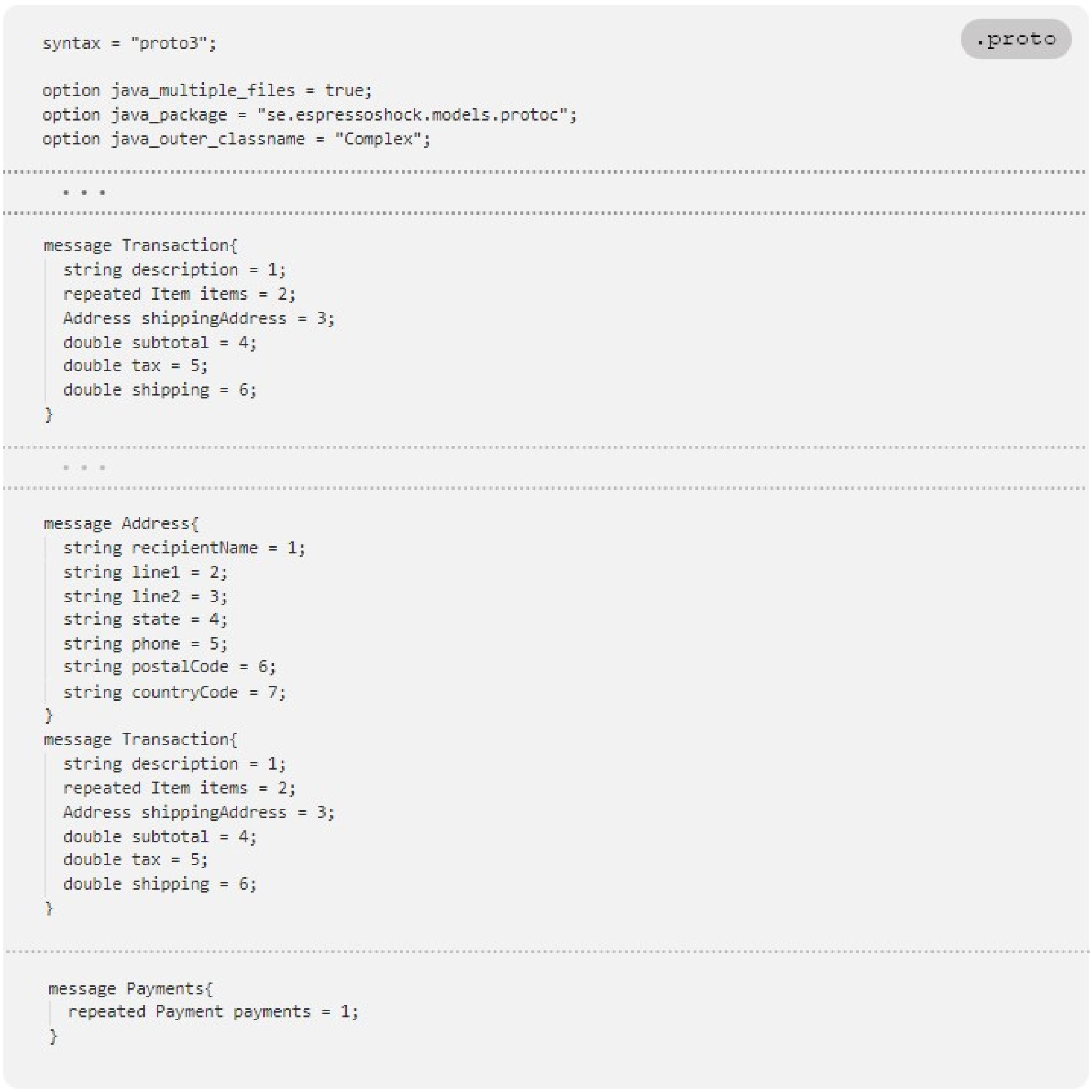


```
json
{
    "users": [
        {
            "uid": "beb1beb2-f69b-4f74-9c3f-83e3a8271739",
            "email": "coleman.balistreri@yahoo.com",
            "username": "vidal.hammes",
            "profilePictureURL": "https://s3.amazonaws.com/uifaces/faces/twitter/mandalareopens/128.jpg",
            "age": 0
        }
    ]
}
```

Figure 21 - User object serialized - JSON response extract with dummy data

4.7.2 Nested Approach

To thoroughly study, compare and ultimately evaluate whatever Google's binary interchange format, *Protobuf*, can be used to enhance inter-services communication in modern, *cloud-native*, *container-first,* REST-based *Quarkus* microservices, a more complex data structure has been implemented. A data structured with 4-level deep nested objects, inspired by PayPal's v1 API [64], has been implemented to fully stress and test the binary and textual representational format and serializer, as shown in **Figure 22** (see **Appendix 4: Source code)**.

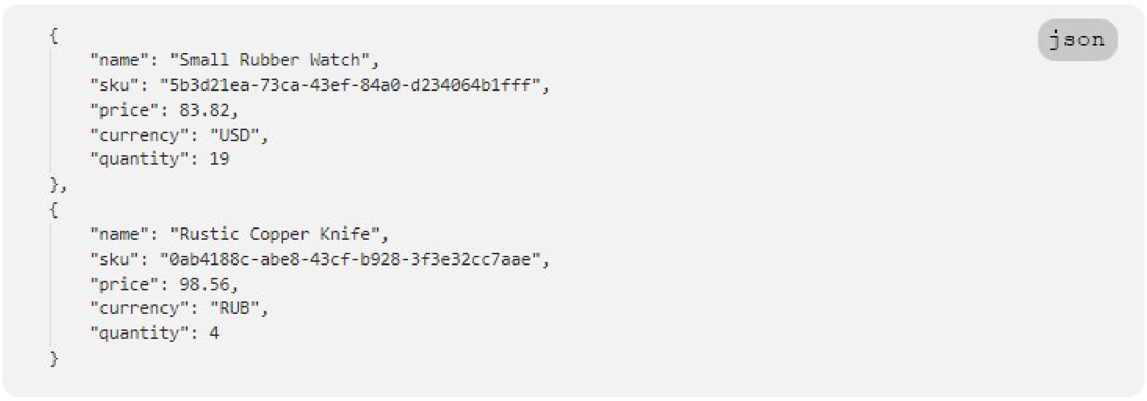


```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "se.espressoshock.models.protoc";
option java_outer_classname = "Complex";

  . . .

message Transaction{
  string description = 1;
  repeated Item items = 2;
  Address shippingAddress = 3;
  double subtotal = 4;
  double tax = 5;
  double shipping = 6;
}

  . . .

message Address{
  string recipientName = 1;
  string line1 = 2;
  string line2 = 3;
  string state = 4;
  string phone = 5;
  string postalCode = 6;
  string countryCode = 7;
}
message Transaction{
  string description = 1;
  repeated Item items = 2;
  Address shippingAddress = 3;
  double subtotal = 4;
  double tax = 5;
  double shipping = 6;
}

message Payments{
  repeated Payment payments = 1;
}
```

Figure 22 – Code segment (extract) of the Proto definition of the Payment nested (complex) data structure

4.8 Dummy Data Generation

Since this study also aims to provide an evaluation of the payload size in respect to each analyzed serializer and serialization format, generating appropriate data, in regards to type and length, during the object construction has been determined to be essential for the accuracy of the study. The data had to be of the right type (string, int, float, etc.) as well as have the appropriate length in order to simulate a real application scenario and to avoid any possible compiler optimization. All the dummy data has been generated using the java port of the popular Ruby's *faker* [65] gem library. An example of the accuracy of the generated data using *faker* is shown in **Figure 23**.



```json
{
    "name": "Small Rubber Watch",
    "sku": "5b3d21ea-73ca-43ef-84a0-d234064b1fff",
    "price": 83.82,
    "currency": "USD",
    "quantity": 19
},
{
    "name": "Rustic Copper Knife",
    "sku": "0ab4188c-abe8-43cf-b928-3f3e32cc7aae",
    "price": 98.56,
    "currency": "RUB",
    "quantity": 4
}
```

Figure 23 - Faker data generation example (JSON Response Extract)

5 Results

5.1 Literature review results

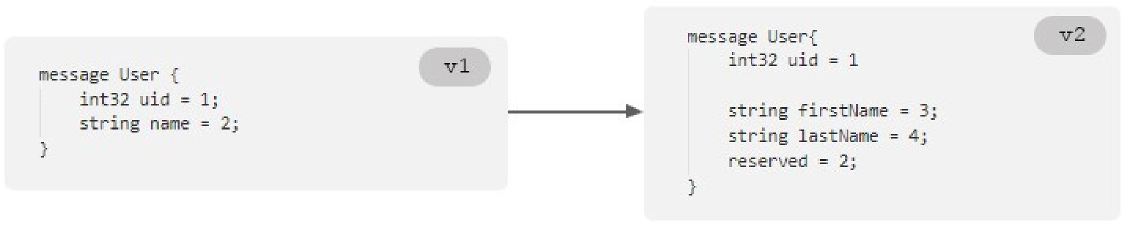
The proposed literature review results have been produced, due to the lack and scarcity of scientific publications and articles, at least as a result of the research conducted upon the authoritative libraries discussed in section **3.1**, using primarily the official documentation provided by the authors of their respective technologies. Moreover, further findings have highlighted the source of the absence present in the research field and the reasoning behind it. The majority of the articles published in our field of interest, in function and regarding the scope of the *research question 2* (see section **1.2**), describe the implication and implementation of data versioning and migration strategies of Google's language-neutral, binary data interchange format, *Protocol Buffers* [2] in the context of gRPC. The explanation can be found and traced back to the fact that both technologies are authored by the same company, Google. In addition, H.Bagci et al. [66] gRPC employs *Protocol buffers* as their serialization format, as also stated in the official gRPC documentation [67]. These results have been excluded as they inherently fall outside the scope of this study. This thesis, as outlined in section **1.3**, analyses the interservice communication that is realized through the use of REST mechanisms and not RPC, as considered previously when discussed regarding gRPC (see **Appendix 1: Microservices**). An example of a typical microservice architecture that utilized both REST mechanism along with gRPC technologies can be seen in **Figure 39.**

Protocol Buffers [2] are designed to be backward and forwards compatible, providing out-of-the-box tools to handle data evolution through an improved system of versioning management that allows updating existing *message type definitions* without breaking the interoperability of systems that utilize legacy or older versions of the *message*. This is realized, per specification, through the use of *Reserved Fields* and by the application of a set of *Rules and Recommendations* [68].

Reserved tags are used to communicate to the developers, as well as to the *protoc* compiler, that these tags are reserved and should not be utilized. This is extremely important for versioning; whereas a *message type definition* is updated by either removing or commenting out a field, future developers might re-use the same field tag. This can cause several issues, including but not limited to data corruption and privacy bugs as they are referring to different *message definitions* [69]. **Figure 24** shows an example of a *message type definition* being updated (the field *name* has been split in *firstName* and *lastName*), and the *reserved tag* is used to handle the data evolution, communicating future developers to not utilized the field with tag 2, as well as telling the *protoc* compiler to prevent compilation if such field would be used. Therefore through documentation as well as tooling support, it has been explicitly shown that the next available tag is 5. *Reserved tags* can be enumerated in multiple ways, allowing single or multiple tags to be reserved in a single line, as illustrated in **Figure 26**.

In scenarios where guaranteeing interoperability with other services that might not support *Protocol Buffers* binary data interchange format is a necessity, *Reserved tags,* despite they may be formally correct, and the *messages* compile successfully, cannot distinguish between context ambiguities. In those cases, *Reserved fields* are used. **Figure 25** illustrates a case scenario where a microservice communicates with a service that does not support *Protocol Buffers*. The *Protobuf* message is then encoded in a textual serialization format such as JSON. Even though in the Protocol Buffer context, in the updated definition of our message (v2), the field *fullName* has a different meaning and interpretation than the one in the first version (v1), when the message is serialized in JSON, the meaning and context are lost. **Figure 42** illustrates the usage of *Reserved fields* over *Reserved tags* to address the previously described issues.

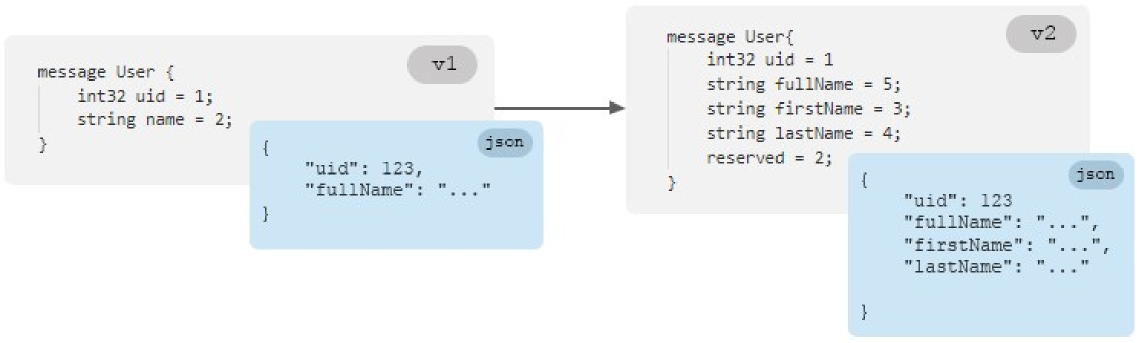
Along with these tools, a set of *Rules and Recommendations* [68] is provided in order to guarantee seamless versioning and data migration. A list with all the rules and recommendations provided by the *Protobuf* authors can be found in Appendix: **Theoretical work**.



```
message User {
    int32 uid = 1;
    string name = 2;
}
```

```
message User{
    int32 uid = 1

    string firstName = 3;
    string lastName = 4;
    reserved = 2;
}
```

Figure 24 - Example scenario of a proto message type definition update - field name is split into firstName and lastName



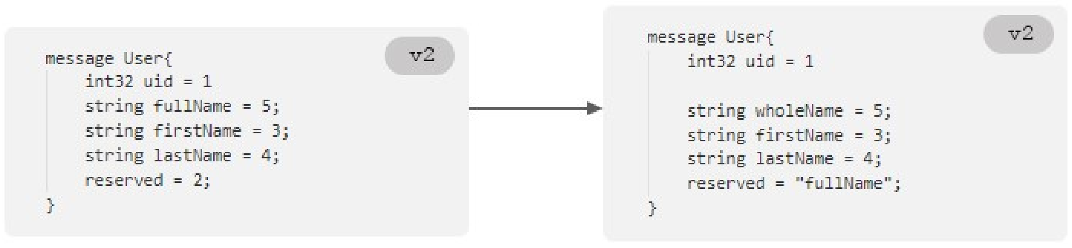
```
message User {
    int32 uid = 1;
    string name = 2;
}
```

```
{
    "uid": 123,
    "fullName": "..."
}
```

```
message User{
    int32 uid = 1
    string fullName = 5;
    string firstName = 3;
    string lastName = 4;
    reserved = 2;
}
```

```
{
    "uid": 123
    "fullName": "...",
    "firstName": "...",
    "lastName": "..."
}
```

Figure 25 - Example scenario of a proto message type definition update encoded to JSON – context is lost



```
message User{
    int32 uid = 1
    string fullName = 5;
    string firstName = 3;
    string lastName = 4;
    reserved = 2;
}
```

```
message User{
    int32 uid = 1

    string wholeName = 5;
    string firstName = 3;
    string lastName = 4;
    reserved = "fullName";
}
```

Figure 26 - Example scenario of a proto message type definition update - to prevent loss of context in case of encoding to textual serialization format, the reserved field is used over the reserved tag

5.2 Empirical results

This section exposes the results obtained from the data gather executing the benchmarks discussed in section **4.4**. The response time benchmarks, together with the request processing times benchmarks, have been executed 100 times sequentially with an initial pre-request in order to obtain accurate data samples. Where applicable, the standard deviation has been calculated and graphed accordingly in the form of error bars. More detailed graphs and charts have also been supplied as supplementary material and can be found in **Appendix 3: Supplementary result materials.**

5.2.1 Response time

The response time of two containerized microservice has been tested through the use of the benchmarks discussed in section **4.4,** and the results have been proposed below. This section analyzes and compares the behavior of the system when a flat data structure and a nested data structure (see section **4.7**) are used.

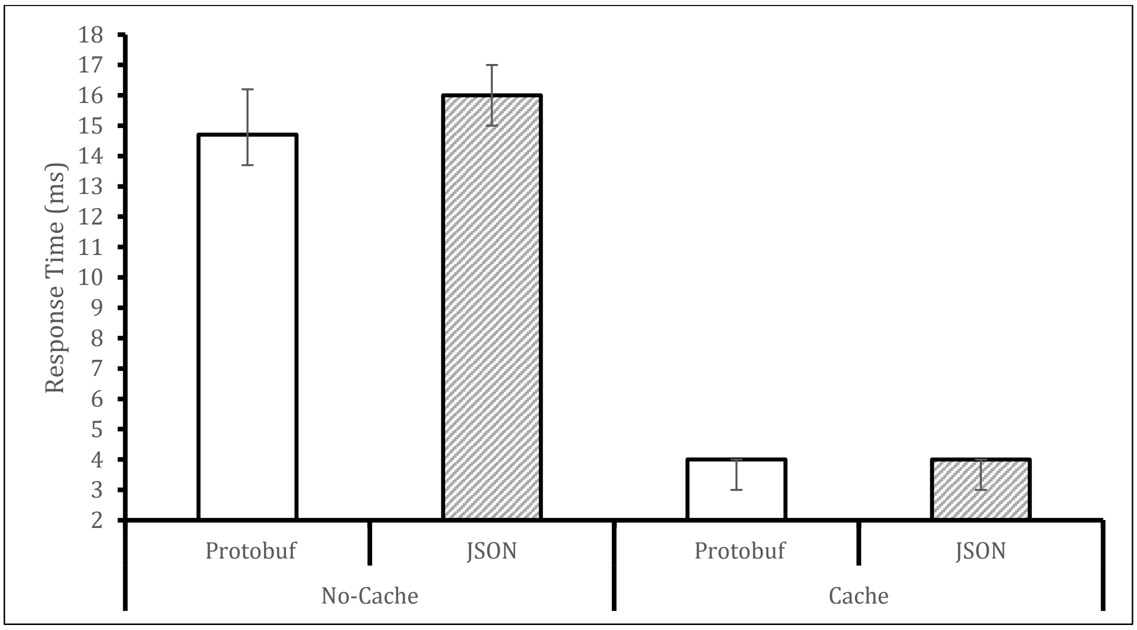


Figure 27 - Response time benchmark results - GET / Data Generation 5 / Uncompressed / cache vs no-cache comparison

5.2.2 Request Processing Time

The *Request Processing Time* of a containerized microservice has been tested through the use of the benchmarks discussed in section 4.3, and the results have been proposed below. The following test has been performed on the nested data structure (see section **4.7.2**) with a data generation count of up to 650 elements (see section **4.4**).

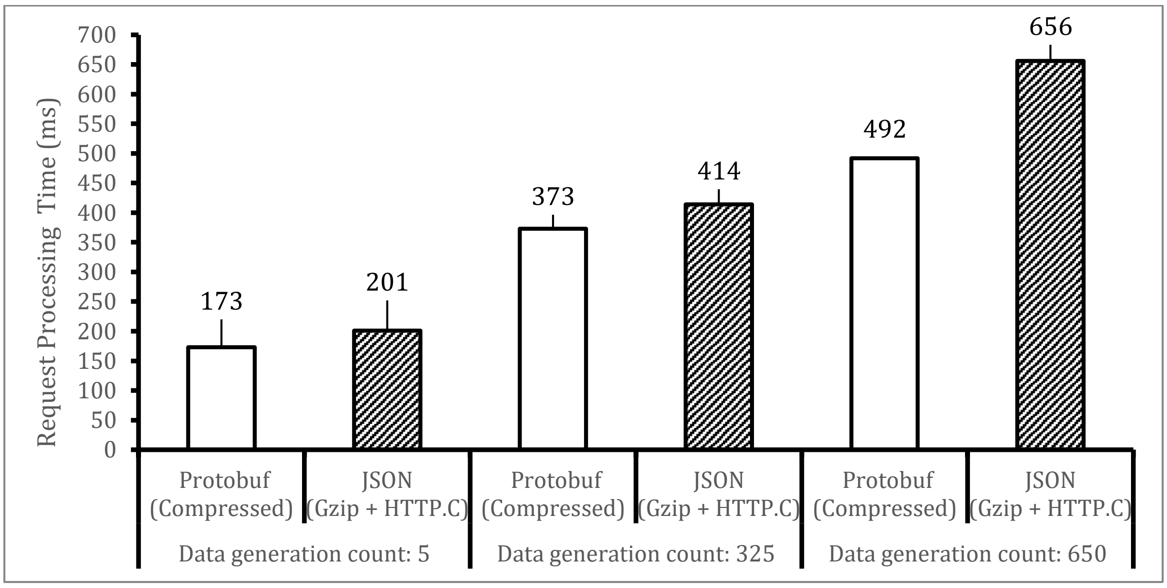


Figure 28- Request Processing Time benchmarks Results / POST /Compressed / no-cache / Data generation count comparison over time

5.2.3 Payload size

The following benchmarks have been designed to test and evaluate the different request and response payload sizes when (I) data structure and complexity (see section **4.7**) increments, and (II) the object count increases.

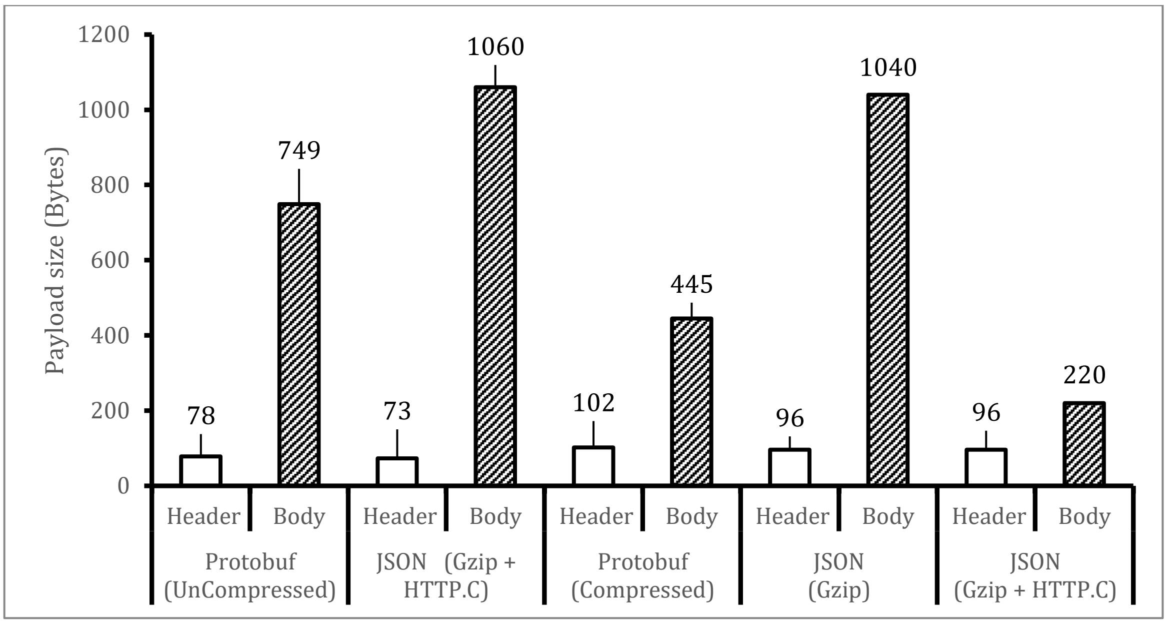


Figure 29 - Paylod size benchmark results / Data generation count 5 / flat data structure / compression comparison

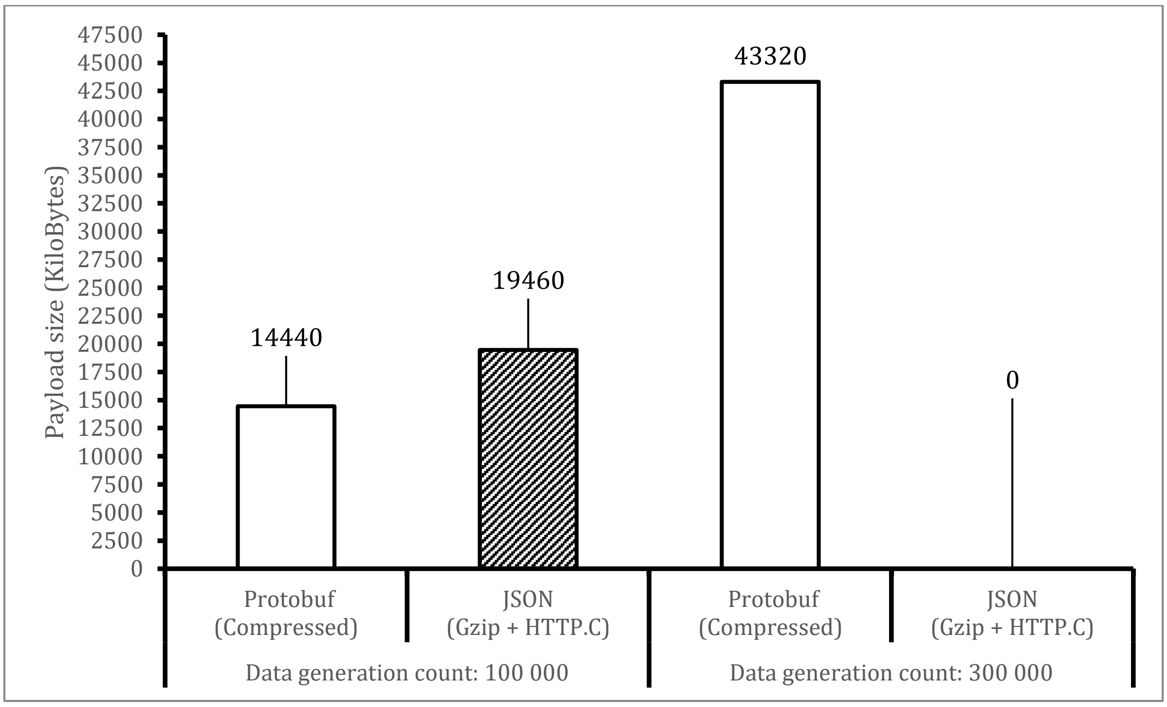


Figure 30 - Paylod size benchmark results / flat data structure / data generation count comparison

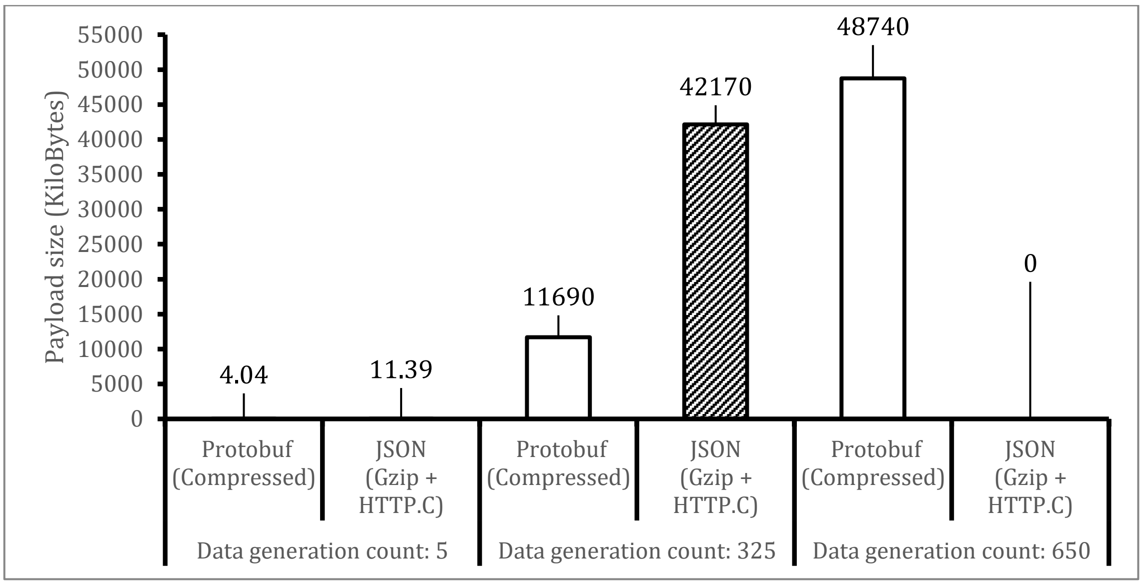


Figure 31 - Paylod size benchmark results / nested data structure / data generation count comparison

5.2.4 Memory analysis

K-native applications are required to provide a small memory footprint in order to reduce cloud costs as well as to provide the necessary performances, demanding every instance to be as small and as efficient as possible. To further reinforce and motivate the reasonings provided in section **6.0**, the memory of each microservices has been studied with the tools described in section **4.3.**

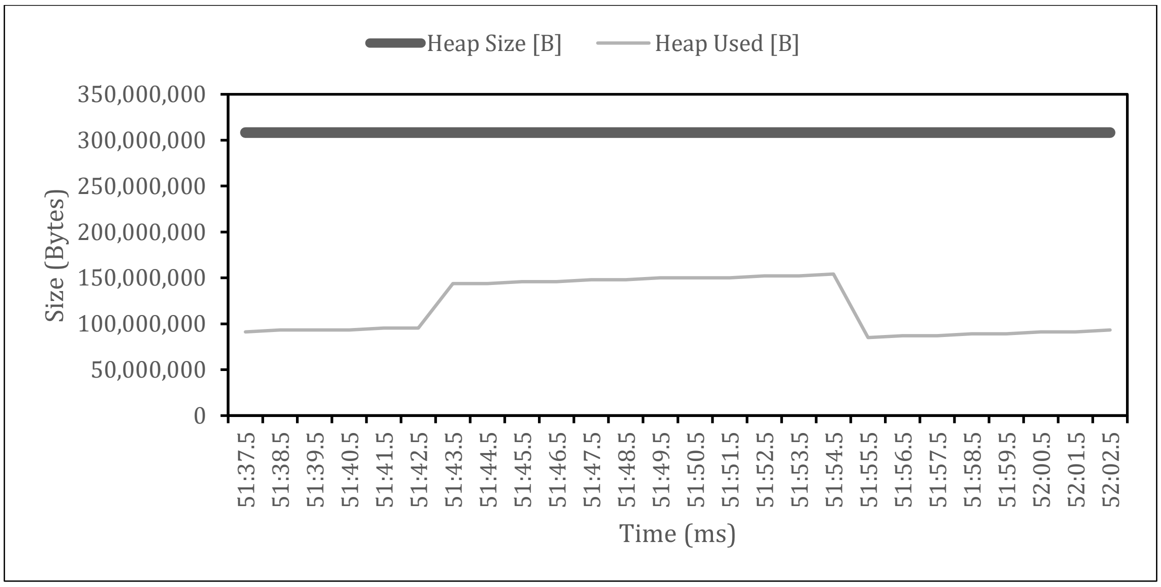


Figure 32 - Memory analysis of the benchmark illustrated in ***Figure 43 during the JSON Serialization***

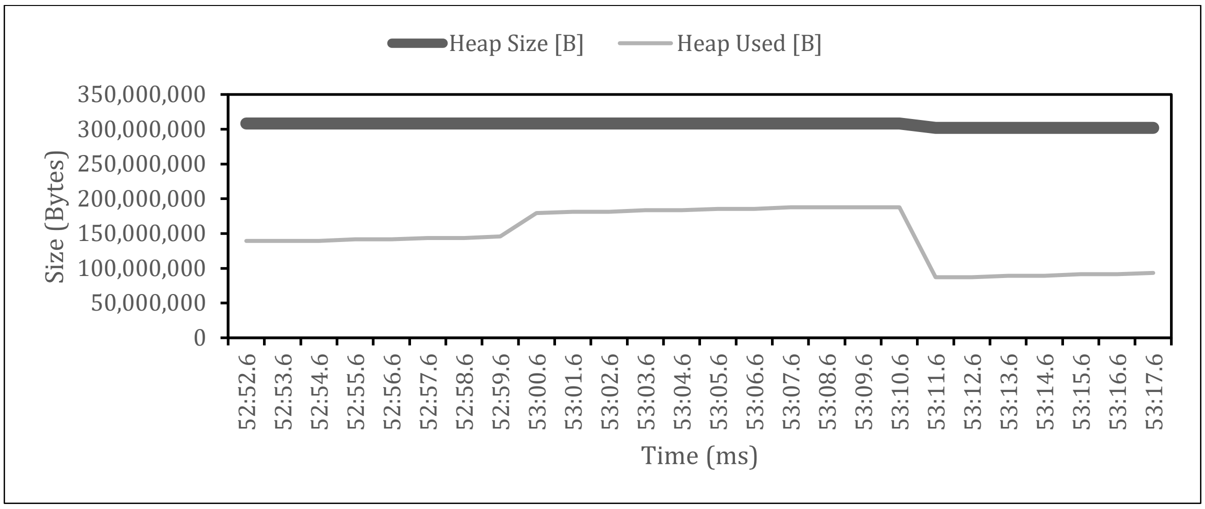


Figure 33 - Memory analysis of the benchmark illustrated in ***Figure 43*** *during the Protobuf Serialization*

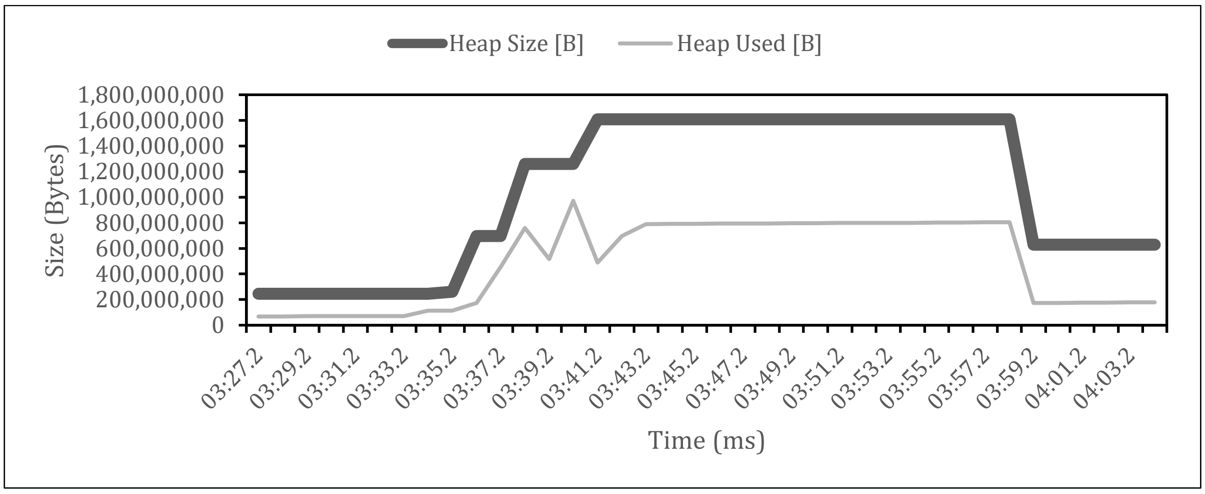


Figure 34 - Memory analysis of the benchmark illustrated in ***Figure 44*** *during the JSON unsuccessful Serialization*

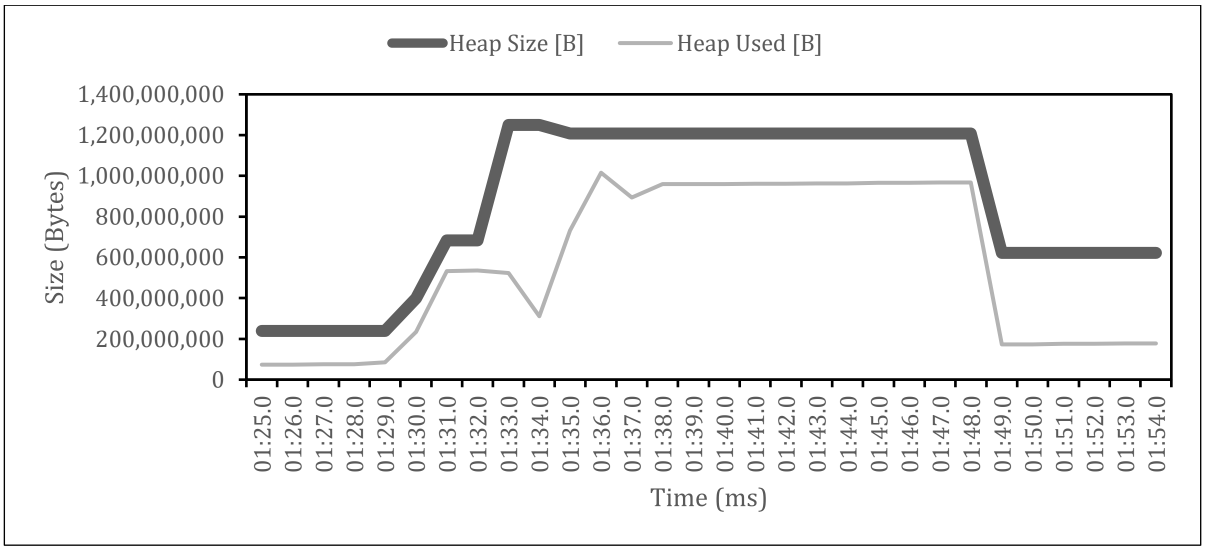


Figure 35 - Memory analysis of the benchmark illustrated in ***Figure 44*** *during the Protobuf Serialization*

6 Discussion

6.1 Research Question 1

This study suggested that a modern, Supersonic, Cloud-Native, K-Native REST-based Java microservice endpoint can be enhanced, from a performance standpoint, by utilizing Google's language-neutral, platform-neutral, binary-based data interchange format over a traditional textual serialization format such as JSON or XML. The findings proposed in section **5.2** suggest that the implementation of *Protobuf* as a binary data interchange format to be utilized across multiple services, through the use of lightweight RESTful mechanism over the HTTP Protocol, does, in fact, lead to performance improvements in regards to *response time, payload size* and *transient memory usage*. However, it does not come without caveats. *Protocol buffers* provide a higher serialization and deserialization speed compared to JSON when serialized with *Jackson*, as shown in **Figure 7** (see **2.3.2 Binary Formats**); however, in the scope of this study, as illustrated in **Figure 27**, the increase in serialization speed is negligible as it leads to only less than 2ms improvements in service's response time corresponding to an 8.13% improvement across the board. It must also be considered that in containerized microservices, where computational power is expensive, caching policies are a very common solution to easily reduce resource consumptions. It can be observed, in Figure 27, that the implementation of caching policies is extremely effective, and in our testing aid to the same response time regarding the serialization format in use, while providing even better metrics than baseline due to the data been cached rather than re-computed. The most notable improvements have been observed during the execution of *Request Processing Time* benchmarks, through the use of POST requests, where the deserialization process was investigated. As previous studies suggested, where benchmarks have been proposed in section **2.3.2**, the major difference measured in execution times between *Protocol buffers* and JSON/*Jackson* has been recorded during the deserialization phase rather than the serialization phase, as JSON appeared to be slower to be *parsed* rather than to *encoded*. In these scenarios, our benchmarks, as discussed in section **5.2.1**, and depicted in **Figure 28**, showed an improvement of 25.1%, with a latency reduction in the best case observed of 164.3ms.

The payload size, on the other hand, according to our test results, gained a considerable improvement due to the binary nature of *Protobuf* when compared to JSON. *Protocol* *buffers* outperformed JSON in almost every one of our tests, providing significantly smaller objects, with the only exception being when serializing very small objects with low objects count as visible in **Figure 29**. This indicates that this is the point where the extra metadata, added by *Protobuf* to later deserialize the data, occupies more space than the advantage gained from the binary representation of the data. The binary format provided, at worst, 25.80% smaller payloads, as shown in **Figure 30** with a best-case scenario of 58.2% smaller serialized objects compared to JSON when only compressed with Gzip, as depicted in **Figure 29**, for flat data structures with a low object count. Moreover, the payload size benchmark results of the nested data structure revealed an improvement over JSON, in the worst-case scenario of 64.54% smaller object's size and a remarkable best-case scenario of 72.28% smaller payloads. It is also necessary to point out, due to the testing design created to provide an *average-case* as well as *edge-case scenarios*, that in every test conducted, JSON wasn't able to serialize the object within the set memory limit requirements; in these cases, the value 0 has been marked accordingly on the provided graphs.

The efficiency of Protocol buffers, when compared to JSON, has been observed to be increasing with the (I) complexity and (II) size of objects. As shown in section **5.2.3**, the higher the complexity, especially in terms of nested objects, of the data structure to be encoded, the higher the difference in system latency has been registered, especially during the deserialization phase, partially for the reasons previously discussed. Further findings, during our memory analysis, discussed in section **5.2.4,** reinforced what our study has suggested; despite the memory usage appears to have a very similar profile, as can be observed in **Figure 32** and **Figure 33**, the *Protobuf* compiler seems to be utilizing memory more efficiently. Interestingly, in **Figure 33**, *Protoc* uses more heap memory than *Jackson*, as shown in **Figure 32**, but still within the 400MB, pre-imposed memory limit. Subsequently, on the contrary, when undergoing a more demanding test, with a more complex and bigger object, that ultimately resulted in a big 43320KB binary blob, as shown in **Figure 30,** it completed the serialization using more transient memory but requesting (allocating) less heap memory allowing it to fully complete the serialization; while *Jackon* allocated more memory and the Docker's memory constraints, killed the process before the serialization was completed.

6.2 Research Question 2

The prerogative and end goal of microservices, as broadly discussed in section **2.0** and in more detail later in section **2.1**, is to design a system as a collection of loosely coupled services that are self-contained and provide only a specific business functionality. Traditional code-heavy monolith applications are, in this new way of designing architectures, break down into independently, more serviceable, deployable microservices that are typically deployed as containers on container clusters [10]. This novel approach of designing systems mitigate, or in some cases, completely eliminated the risk of single-point-of-failure; however, in an architecture where multiple applications have to communicate and cooperate to produce a result, new communications challenges arise. One of the most notable and historical issues that have been afflicting microservices since their novel implementation has been the raw material that microservices work with and manage: data. Due to its intrinsic nature, data is subjected to change and be updated; this is especially true in RESTful APIs and microservices, where a single service can unexpectedly change its interface and inadvertently breaks the contract that it prior established with its consumers. In professional enterprise scenarios, as discussed in this thesis, being able to proactively react to data changes and evolutions by implementing means of versioning management that allows your API to evolve with backward compatibility is essential. The analyzed data interchange format, *Protobuf*, provides such functionalities out-of-box [68]. The built-in message-type definition also allows validating incoming and outgoing messages against a specific *.proto* definition, as discussed in section **2.4**. In professional environments, where system robustness and type-safe are a necessity, messages have to be *strongly typed* to ensure a smooth experience and eliminate unexpected bugs and reduce the risks of exploits at runtime.

7 Conclusion

The study showed promising results regarding the implementation of Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data in modern, *cloud-native*, *container-first,* REST-based *Quarkus* [1] microservice architectures. The empirical results obtained from our testing confirmed our hypothesis formulated in the research questions about the effectiveness of a binary-based serialization format in reducing the response time and payload size, even if not at the expected degree. This research provided a novel insight into the application of *Protocol Buffers* in containerized Quarkus microservices that was not present in the research field prior.

The gathered findings can be summarized, in regards to the research questions initially formulated, as follows:

R1: *From a performance standpoint, how can the response time and payload size be reduced in a modern, Supersonic, Cloud-Native, K-Native REST-based Java Microservice endpoint?*

The response time and payload size can be reduced by utilizing Google's binary data interchange format rather than traditional textual serialization formats.

R2: *How can a strongly typed binary-based representational data interchange format with schema support improve versioning management, data evolution, and migration in microservices?*

Protocol Buffers provide an out-of-the-box mechanism to update the message type definitions defined in *.proto* files. This allows easy message versioning as a means to react to data evolution with backward compatibility as well as data migration.

7.1 Future Work

As briefly discussed in section **1.4**, future work on this subject can be done by:

1. Deploy the benchmarks on online K8s clusters, such as the IBM Cloud Kubernetes Service [70], in order to obtain more accurate data as well as to investigate platform-specific service optimizations.
2. Compare other's binary data interchange formats, as disclosed in section **2.3.2.**

8 References

1. Red Hat Software. Quarkus - Supersonic Subatomic Java [Internet]. [cited 2021 Feb 7]. Available from: https://quarkus.io/

2. Google Inc. Protocol Buffers: Defining A Message Type | Google Developers [Internet]. [cited 2021 Apr 7]. Available from: https://developers.google.com/protocol-buffers

3. Red Hat Software. quarkusio/quarkus: Quarkus: Supersonic Subatomic Java. [Internet]. [cited 2021 May 11]. Available from: https://github.com/quarkusio/quarkus

4. Red Hat Software. Why Kubernetes native instead of cloud native? | Red Hat Developer [Internet]. [cited 2021 May 11]. Available from: https://developers.redhat.com/blog/2020/04/08/why-kubernetes-native-instead-of-cloud-native

5. Saleem R. Cloud Computing's Effect on Enterprises [Internet]. Lund University; 2011. Available from: https://lup.lub.lu.se/student-papers/search/publication/1764306

6. Alshuqayran N, Ali N, Evans R. A Systematic Mapping Study in Microservice Architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). 2016. p. 44–51. , DOI: 10.1109/SOCA.2016.15, ISBN: 978-1-5090-4782-6

7. James Lewis MF. Microservices [Internet]. 2014 [cited 2021 May 7]. Available from: https://martinfowler.com/articles/microservices.html

8. Sam Newman. Building Microservices [Internet]. 2nd ed. O'Reilly Media, Inc.; 2021 [cited 2021 May 7]. Available from: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/, ISBN: 9781492034025

9. Dhalla HK. A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP. In: 16th International Conference on Network and Service Management, CNSM 2020, 2nd International Workshop on Analytics for Service and Application Management, AnServApp 2020 and 1st

International Workshop on the Future Evolution of Internet Protocols, IPFutu. Institute of Electrical and Electronics Engineers Inc.; 2020. , DOI: 10.23919/CNSM50824.2020.9269101

10. Kratzke N, Quint PC. Ppbench a visualizing network benchmark for microservices. In: CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science. SciTePress; 2016. p. 223–31. , DOI: 10.5220/0005732202230231, ISBN: 9789897581823

11. Feitosa Pacheco V. Chained Microservice Design Pattern: Understanding the pattern. In: Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices. Birmingham, England: Packt Publishing; 2018. p. 333. , ISBN: 9781788474030

12. Anithakumari S, Chandrasekaran K. Monitoring and Management of Service Level Agreements in Cloud Computing. Proc - 2015 Int Conf Cloud Auton Comput ICCAC 2015. 2015;204–7. , DOI: 10.1109/ICCAC.2015.28, ISBN: 0769556361

13. Schulz F. Towards measuring the degree of fulfillment of service level agreements. ICIC 2010 - 3rd Int Conf Inf Comput. 2010;3(3):273–6. , DOI: 10.1109/ICIC.2010.254, ISBN: 9780769540474

14. Marchioni F. Hands-On Cloud-Native Applications with Java and Quarkus: Build high performance, Kubernetes-native Java serverless applications. Birmingham, England: Packt Publishing; 2019. , ISBN: 9781838821470

15. Oracle Corporation. GraalVM [Internet]. [cited 2021 Mar 12]. Available from: https://www.graalvm.org/

16. Kind K8s - Kubernetes. kind – Getting Started [Internet]. [cited 2021 Mar 11]. Available from: https://kind.sigs.k8s.io/docs/contributing/getting-started/

17. The Kubernetes Authors. Kind K8s - Kubernetes [Internet]. [cited 2021 Mar 11]. Available from: https://kind.sigs.k8s.io/

18. Pahl C, Jamshidi P. Microservices: A systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER. SciTePress; 2016. p. 137–46. , DOI:

10.5220/0005785501370146, ISBN: 9789897581823

19. Lin B, Chen Y, Chen X, Yu Y. Comparison between JSON and XML in Applications Based on AJAX. Proc - 2012 Int Conf Comput Sci Serv Syst CSSS 2012. 2012;(February 1998):1174–7. , DOI: 10.1109/CSSS.2012.297, ISBN: 9780769547190

20. Maryanne Ndungu. ADOPTION OF THE MICROSERVICE ARCHITECTURE [Internet]. ÅBO AKADEMI; 2019. Available from: http://urn.fi/URN:NBN:fi-fe2019040310910, DOI: http://urn.fi/URN:NBN:fi-fe2019040310910

21. Pham NM. A proposal for a cloud-based microservice architecture for the Skolrutiner system [Internet]. Uppsala University; 2020. Available from: http://www.teknat.uu.se/student, DOI: urn:nbn:se:uu:diva-428348

22. Cloud Native Computing Foundation. Kubernetes [Internet]. [cited 2021 Mar 21]. Available from: https://kubernetes.io/

23. Soto A, Porter J. Quarkus Cookbook: Kubernetes-Optimized Java Solutions. Sebastopol, CA: O'Reilly Media; 2020. , ISBN: 9781492062653

24. Red Hat Software. Quarkus reduces boot time and memory consumption for container-native apps [Internet]. 2019 [cited 2021 Apr 1]. Available from: https://www.redhat.com/en/resources/quarkus-infographic-supersonic-subatomic-java

25. Oracle Corporation. Polyglot Programming [Internet]. [cited 2021 Feb 24]. Available from: https://www.graalvm.org/reference-manual/polyglot-programming/

26. Oracle Corporation. Truffle Language Implementation Framework [Internet]. [cited 2021 Feb 24]. Available from: https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/

27. Oracle Corporation. Quarkus: Container First [Internet]. [cited 2021 Mar 21]. Available from: https://quarkus.io/vision/container-first

28. Oracle Corporation. Quarkus - Contexts and Dependency Injection [Internet]. [cited 2021 Apr 28]. Available from: https://quarkus.io/guides/cdi-reference

29. Oracle Corporation. Oracle: Using Java Reflection [Internet]. [cited 2021 Apr 27].

Available from: https://www.oracle.com/technical-resources/articles/java/javareflection.html

30. Red Hat Software. Quarkus - Building a Native Executable [Internet]. [cited 2021 Apr 27]. Available from: https://quarkus.io/guides/building-native-image

31. Sipek M, Muharemagic D, Mihaljevic B, Radovan A. Enhancing performance of cloud-based software applications with GraalVM and quarkus. In: 2020 43rd International Convention on Information, Communication and Electronic Technology, MIPRO 2020 - Proceedings. 2020. p. 1746–51. , DOI: 10.23919/MIPRO48935.2020.9245290, ISBN: 9789532330991

32. Emeakaroha VC, Healy P, Fatema K, Morrison JP. Analysis of Data Interchange Formats for Interoperable and Efficient Data Communication in Clouds. In: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing. 2013. p. 393–8. , DOI: 10.1109/UCC.2013.79, ISBN: 978-0-7695-5152-4

33. Nurzhan Nurseitov, Michael Paulson, Randall Reynolds CI. Comparison of JSON and XML Data Interchange Formats: A Case Study. In: Proceedings of the ISCA 22nd International Conference on Computer Applications in Industry and Engineering, CAINE 2009, November 4-6, 2009, Hilton San Francisco Fisherman's Wharf, San Francisco, California, USA [Internet]. ISCA; 2009. p. 157–62. Available from: https://dblp.uni-trier.de/rec/conf/caine/NurseitovPRI09.html?view=bibtex

34. Maeda K. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In: 2012 2nd International Conference on Digital Information and Communication Technology and its Applications, DICTAP 2012. 2012. p. 177–82. , DOI: 10.1109/DICTAP.2012.6215346, ISBN: 9781467307338

35. Sumaray A, Makki SK. A comparison of data serialization formats for optimal efficiency on a mobile platform. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, ICUIMC'12. 2012. , DOI: 2184751.2184810, ISBN: 9781450311724

36. Emeakaroha VC, Healy P, Fatema K, Morrison JP. Analysis of data interchange formats for interoperable and efficient data communication in clouds. In: Proceedings - 2013 IEEE/ACM 6th International Conference on Utility and Cloud

Computing, UCC 2013. 2013. p. 393–8. , DOI: 10.1109/UCC.2013.79, ISBN: 9780769551524

37. Eishay Smith. JVM Serializer comparison - GitHub [Internet]. [cited 2021 Mar 8]. Available from: https://github.com/eishay/jvm-serializers/wiki

38. Kaur G, Fuad MM. An evaluation of protocol buffer. Conf Proc - IEEE SOUTHEASTCON. 2010;459–62. , DOI: 10.1109/SECON.2010.5453828, ISBN: 9781424458530

39. Language Guide (proto3) | Protocol Buffers | Google Developers [Internet]. [cited 2021 May 12]. Available from: https://developers.google.com/protocol-buffers/docs/proto3#updating

40. Google Inc. Encoding | Protocol Buffers | Google Developers [Internet]. [cited 2021 Mar 9]. Available from: https://developers.google.com/protocol-buffers/docs/encoding

41. Nazar Hussain. Encoding Base128 Varints, Explained | Hacker Noon [Internet]. [cited 2021 Mar 9]. Available from: https://hackernoon.com/encoding-base128-varints-explained-371j3uz8

42. Li Y, Lin SJ. Removing Redundancy in Little-Endian Base 128 and the Efficient Decoding Approach. IEEE Commun Lett. 2020;24(11):2411–5. , DOI: 10.1109/LCOMM.2020.3008425

43. Martin Kleppmann. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems [Internet]. 1st ed. Sebastopol, CA: O'Reilly Media; 2017 [cited 2021 May 16]. Available from: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/, ISBN: 9781449373320

44. IEEE. IEEE Xplore [Internet]. [cited 2021 Mar 10]. Available from: https://ieeexplore.ieee.org/Xplore/home.jsp

45. Science and Technology Publications L. SciTePress - SCIENCE AND TECHNOLOGY PUBLICATIONS [Internet]. [cited 2021 Mar 10]. Available from: https://www.scitepress.org/HomePage.aspx

46. Google Inc. Google Scholar [Internet]. [cited 2021 Mar 10]. Available from:

https://scholar.google.com/

47. Association for Computing Machinery. ACM Digital Library [Internet]. [cited 2021 Mar 10]. Available from: https://dl.acm.org/

48. Kristianstad University. Biblioteket Högskolan Kristianstad | HKR.se [Internet]. [cited 2021 Mar 10]. Available from: https://www.hkr.se/biblioteket

49. The gzip home page [Internet]. [cited 2021 Mar 10]. Available from: https://www.gzip.org/

50. The Kubernetes Authors. kubernetes-sigs/kind: Kubernetes IN Docker - local clusters for testing Kubernetes [Internet]. [cited 2021 Mar 11]. Available from: https://github.com/kubernetes-sigs/kind

51. The Kubernetes Authors. kubectl | Kubernetes [Internet]. [cited 2021 Mar 11]. Available from: https://kubernetes.io/docs/tasks/tools/

52. Docker I. What is a Container? | App Containerization | Docker [Internet]. [cited 2021 Mar 11]. Available from: https://www.docker.com/resources/what-container

53. John O'Hara. Quarkus Runtime Performance [Internet]. [cited 2021 Apr 12]. Available from: https://quarkus.io/blog/runtime-performance/

54. Red Hat Software. Quarkus - Tips for writing native applications [Internet]. [cited 2021 Apr 12]. Available from: https://quarkus.io/guides/writing-native-applications-tips

55. Google Inc. GoogleContainerTools/jib: ? Build container images for your Java applications. [Internet]. [cited 2021 Mar 11]. Available from: https://github.com/GoogleContainerTools/jib

56. Red Hat Software. Quarkus - Container Images [Internet]. [cited 2021 May 11]. Available from: https://quarkus.io/guides/container-image

57. GraalVM Developers. graalvm/mandrel: Mandrel is a downstream distribution of the GraalVM community edition. Mandrel's main goal is to provide a native-image release specifically to support Quarkus. [Internet]. [cited 2021 Apr 12]. Available from: https://github.com/graalvm/mandrel

58. Team TP. Postman | The Collaboration Platform for API Development [Internet]. Available from: https://www.postman.com/

59. The Apache Software Foundation. Apache JMeter - Apache JMeter^{TM} [Internet]. Available from: https://jmeter.apache.org/

60. Oracle Corporation. OpenJDK - JConsole [Internet]. Available from: https://openjdk.java.net/tools/svc/jconsole/

61. Jiri Sedlacek TH. VisualVM: Download [Internet]. Available from: https://visualvm.github.io/download.html

62. FasterXML. FasterXML/jackson: Main Portal page for the Jackson project [Internet]. [cited 2021 May 12]. Available from: https://github.com/FasterXML/jackson

63. Red Hat Software. Quarkus - Writing JSON REST Services [Internet]. [cited 2021 Feb 11]. Available from: https://quarkus.io/guides/rest-json#gzip-support

64. Paypal Inc. Payments [Internet]. [cited 2021 Apr 11]. Available from: https://developer.paypal.com/docs/api/payments/v1/

65. DiUS Computing Pty Ltd. DiUS/java-faker: Brings the popular ruby faker gem to Java [Internet]. [cited 2021 Apr 11]. Available from: https://github.com/DiUS/java-faker

66. Bagci H, Kara A. A lightweight and high performance remote procedure call framework for cross platform communication. In: ICSOFT 2016 - Proceedings of the 11th International Joint Conference on Software Technologies. SciTePress; 2016. p. 117–24. , DOI: 10.5220/0005931201170124, ISBN: 9789897581946

67. gRPC Authors. Introduction to gRPC | gRPC [Internet]. [cited 2021 Apr 15]. Available from: https://grpc.io/docs/what-is-grpc/introduction/

68. Google Inc. Updating A Message Type | Protocol Buffers | Google Developers. Available from: https://developers.google.com/protocol-buffers/docs/proto3#updating

69. Google Inc. Reserved Fields | Protocol Buffers | Google Developers [Internet]. Available from: https://developers.google.com/protocol-buffers/docs/overview#reserved

70. IBM. IBM Cloud Kubernetes Service - Overview - Sweden | IBM [Internet]. [cited 2021 May 12]. Available from: https://www.ibm.com/se-en/cloud/kubernetes-

service

71. Charlotte Mach CL. What Is Inter-Process Communication? [Internet]. 2021 [cited 2021 May 16]. Available from: https://blog.container-solutions.com/wtf-is-inter-process-communication

72. Buono Vincenzo. espressoshock/da399b-supplementary-material-public: Enhance Inter-service Communication in Supersonic K-Native REST-based Java Microservice Architectures - Supplementary material and source code [Internet]. [cited 2021 May 16]. Available from: https://github.com/espressoshock/da399b-supplementary-material-public

Appendices

Appendix 1: Microservices

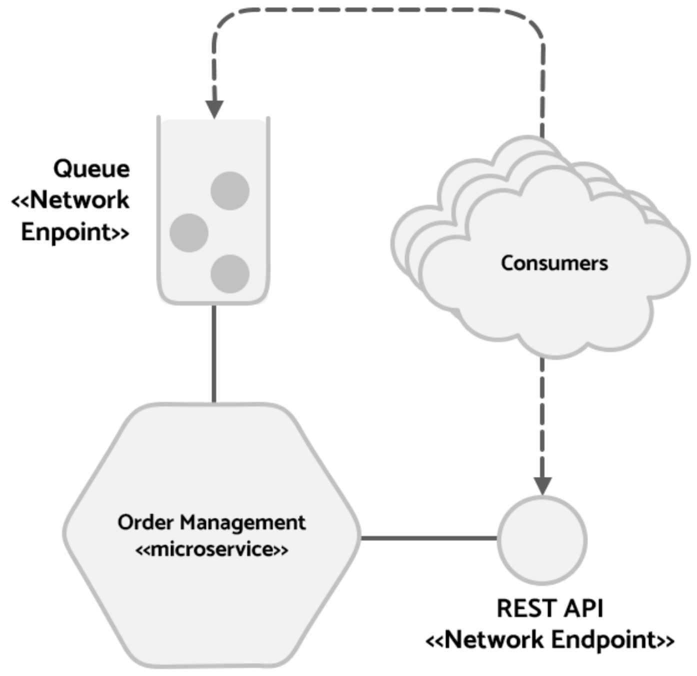


Figure 36 - A microservice exposing its functionality over a REST API and a queue [8]

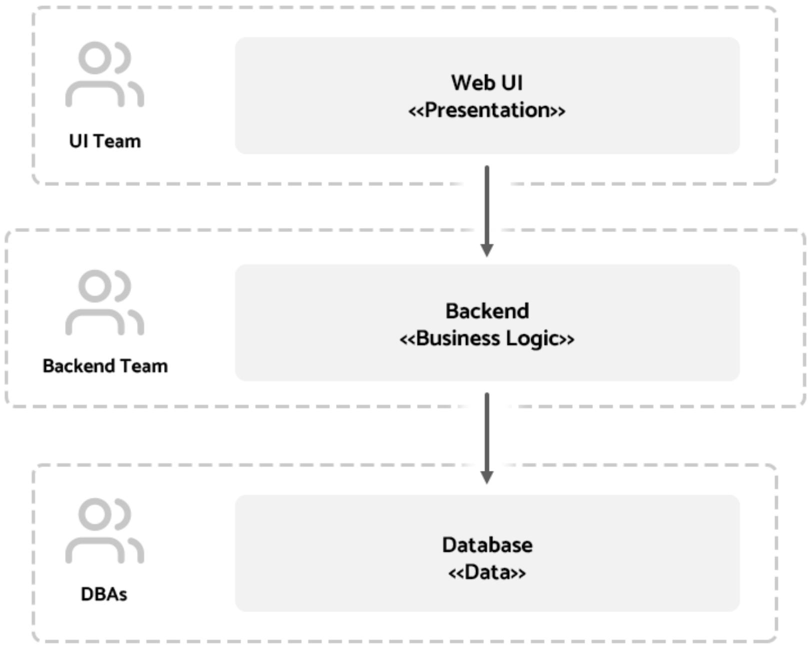


Figure 37 - A traditional three-tiered architecture [8]

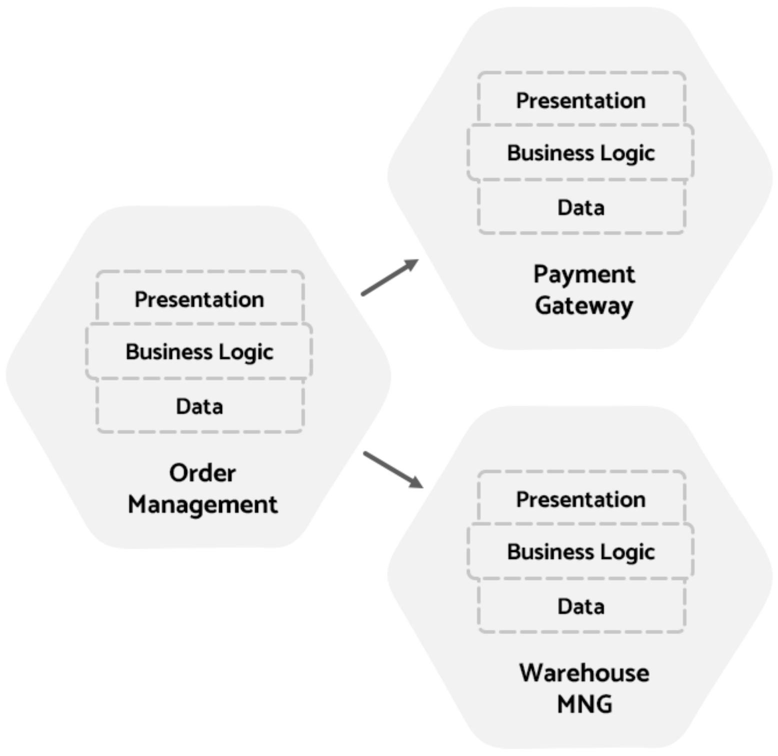


Figure 38 - Each microservice, if required, can encapsulate presentation, business logic, and data storage functionality [8]

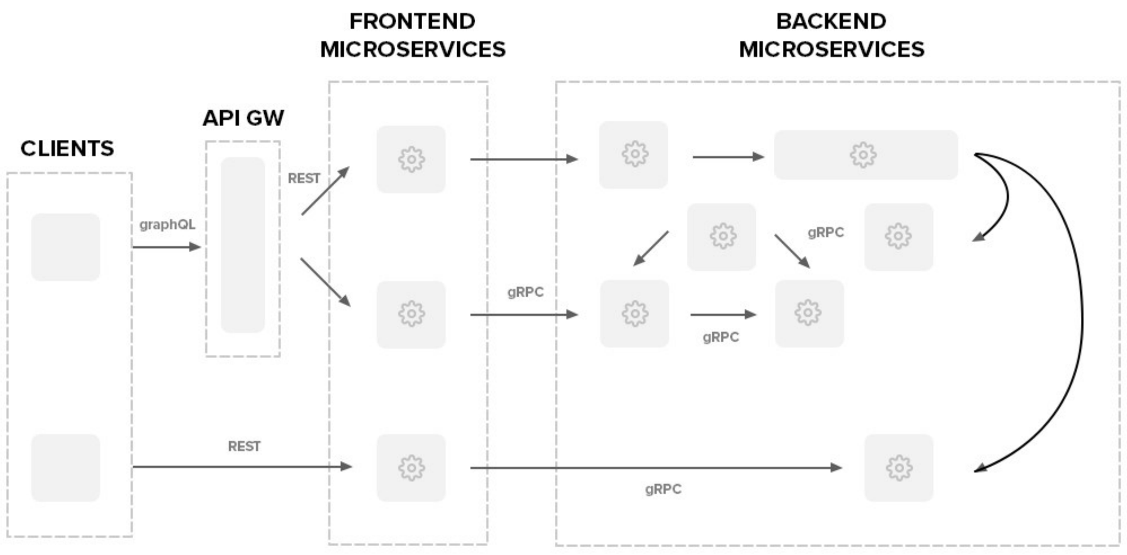
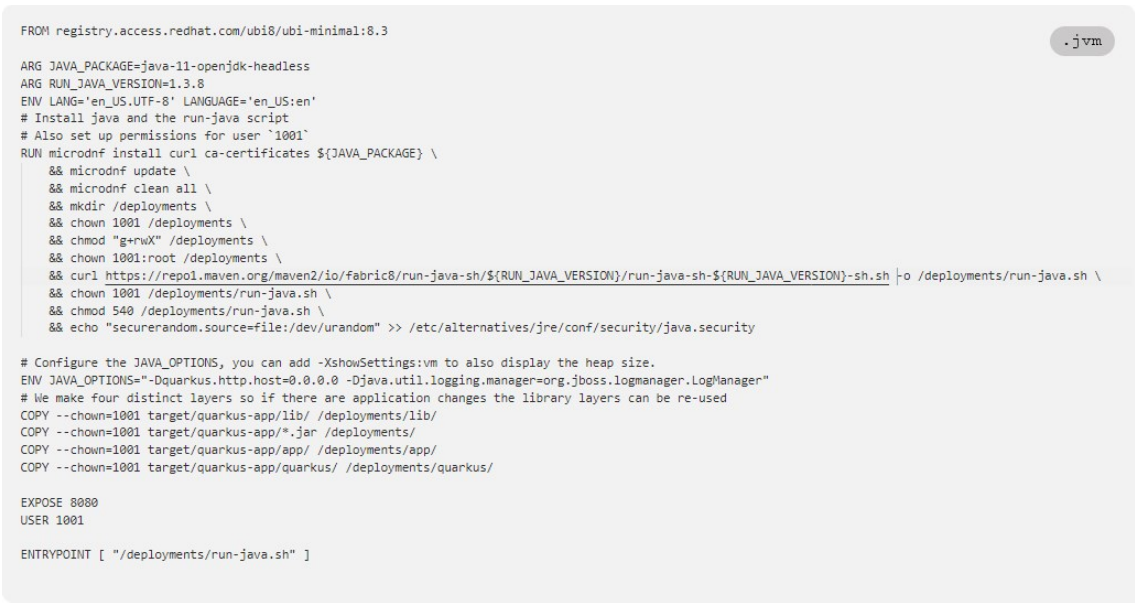


Figure 39 - Example of Microservice architecture that bundles front-end and microservices with API gateways, REST and gRPC [71]

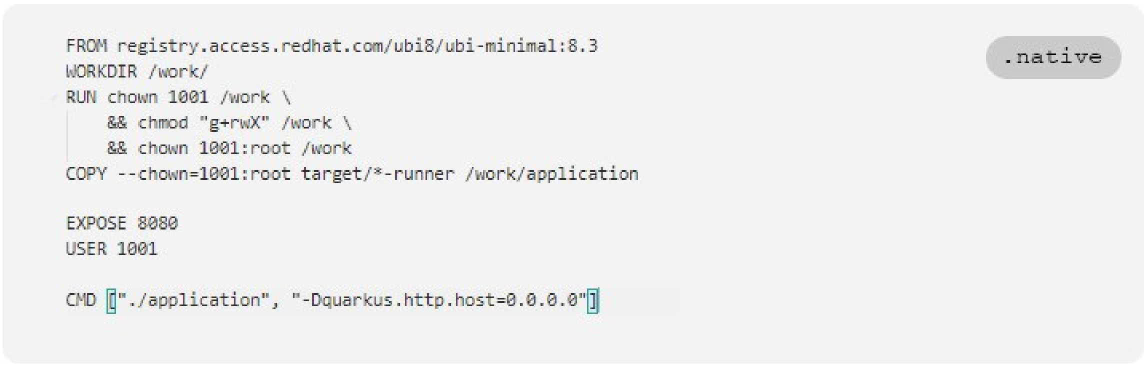
Appendix 2: Docker build

This section contains the files used for the Docker build. The full files in their integrity are also provided in this section. **Figure 40** shows the content of the file *Dockerfile.jvm* while **Figure 41** illustrates the content of the file *Dockerfile.native* used for building the native image.



```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.3

ARG JAVA_PACKAGE=java-11-openjdk-headless
ARG RUN_JAVA_VERSION=1.3.8
ENV LANG='en_US.UTF-8' LANGUAGE='en_US:en'
# Install java and the run-java script
# Also set up permissions for user `1001`
RUN microdnf install curl ca-certificates ${JAVA_PACKAGE} \
    && microdnf update \
    && microdnf clean all \
    && mkdir /deployments \
    && chown 1001 /deployments \
    && chmod "g+rwX" /deployments \
    && chown 1001:root /deployments \
    && curl https://repo1.maven.org/maven2/io/fabric8/run-java-sh/${RUN_JAVA_VERSION}/run-java-sh-${RUN_JAVA_VERSION}-sh.sh -o /deployments/run-java.sh \
    && chown 1001 /deployments/run-java.sh \
    && chmod 540 /deployments/run-java.sh \
    && echo "securerandom.source=file:/dev/urandom" >> /etc/alternatives/jre/conf/security/java.security

# Configure the JAVA_OPTIONS, you can add -XshowSettings:vm to also display the heap size.
ENV JAVA_OPTIONS="-Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager"
# We make four distinct layers so if there are application changes the library layers can be re-used
COPY --chown=1001 target/quarkus-app/lib/ /deployments/lib/
COPY --chown=1001 target/quarkus-app/*.jar /deployments/
COPY --chown=1001 target/quarkus-app/app/ /deployments/app/
COPY --chown=1001 target/quarkus-app/quarkus/ /deployments/quarkus/

EXPOSE 8080
USER 1001

ENTRYPOINT [ "/deployments/run-java.sh" ]
```

Figure 40 - NVM-mode docker file



```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.3
WORKDIR /work/
RUN chown 1001 /work \
    && chmod "g+rwX" /work \
    && chown 1001:root /work
COPY --chown=1001:root target/*-runner /work/application

EXPOSE 8080
USER 1001

CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```

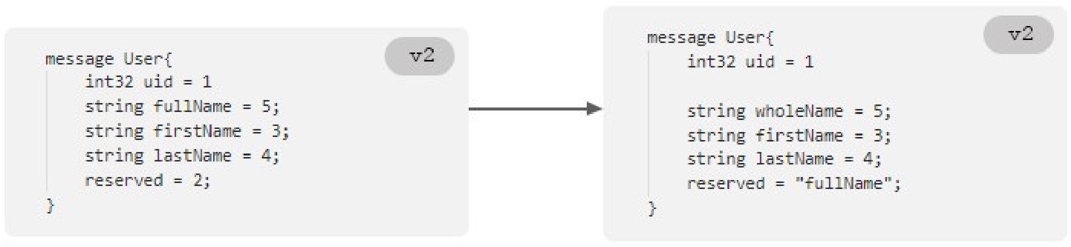
Figure 41 - Native-mode docker file

Appendix 3: Supplementary result materials

This appendix reports supplementary results materials, including but not limited to graphs, charts, tables, and figures.

Theoretical work

This section contains supplementary material related from the conducted literature review. **Figure 42** shows an example of the usage of the *reserved field* over the *reserved tag* to preserve the context.



```
message User{                         v2
    int32 uid = 1
    string fullName = 5;
    string firstName = 3;
    string lastName = 4;
    reserved = 2;
}
```

```
message User{                         v2
    int32 uid = 1

    string wholeName = 5;
    string firstName = 3;
    string lastName = 4;
    reserved = "fullName";
}
```

Figure 42 - Example scenario of a proto message type definition update - to prevent loss of context in case of encoding to textual serialization format, the reserved field is used over the reserved tag

Table 4 - Rules and Recommendations provided by the Protobuf Authors when updating a message type definition [68]

Recommendation No.	Description
0	Don't change the field numbers for any existing fields.
1	Any new fields that you add should be optional or repeated
2	Non-required fields can be removed, as long as the field number is not used again in your updated message type. You may want to rename the field instead, perhaps adding the prefix "OBSOLETE_",
3	A non-required field can be converted to an extension and vice versa, as long as the type and number stay the same
4	int32, uint32, int64, uint64, and bool are all compatible
5	sint32 and sint64 are compatible with each other but are not compatible with the other integer types.
6	string and bytes are compatible as long as the bytes are valid UTF-8.
7	Embedded messages are compatible with bytes if the bytes contain an encoded version of the message.
8	fixed32 is compatible with sfixed32, and fixed64 with sfixed64
9	For string, bytes, and message fields, optional is compatible with repeated

Empirical Experiment

This section provides supplementary graphs of the conducted experiment, including but not limited to, the request and response header and body size. Figure 43 depicts benchmark results regarding the payload size with a data generation count of 100 000 elements while **Figure 44** shows the payload size when the data count is increased to 300 000 elements. **Figure 45** illustrates the payload size when nested objects are serialized and presents a comparison of the analyzed data interchange formats. **Figure 46** represents the same benchmark but with a higher element count (325), while **Figure 47** compares the latter when the data generation count is set to 650 elements.

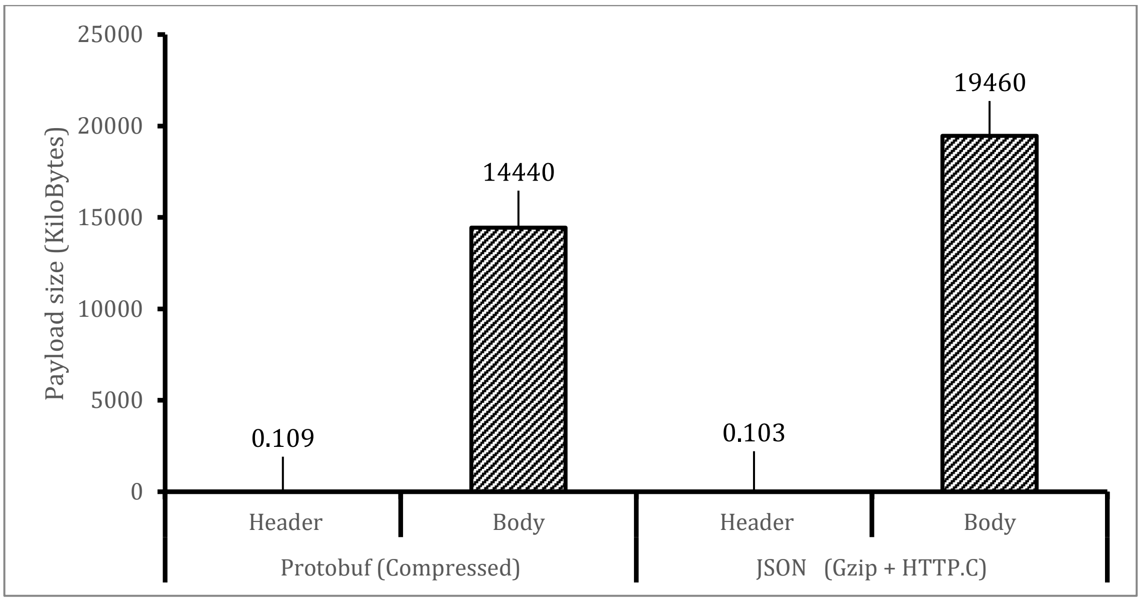


Figure 43 - Payload size benchmark results / Data Generation Count 100 000 / flat data structure / data presented with header and body size

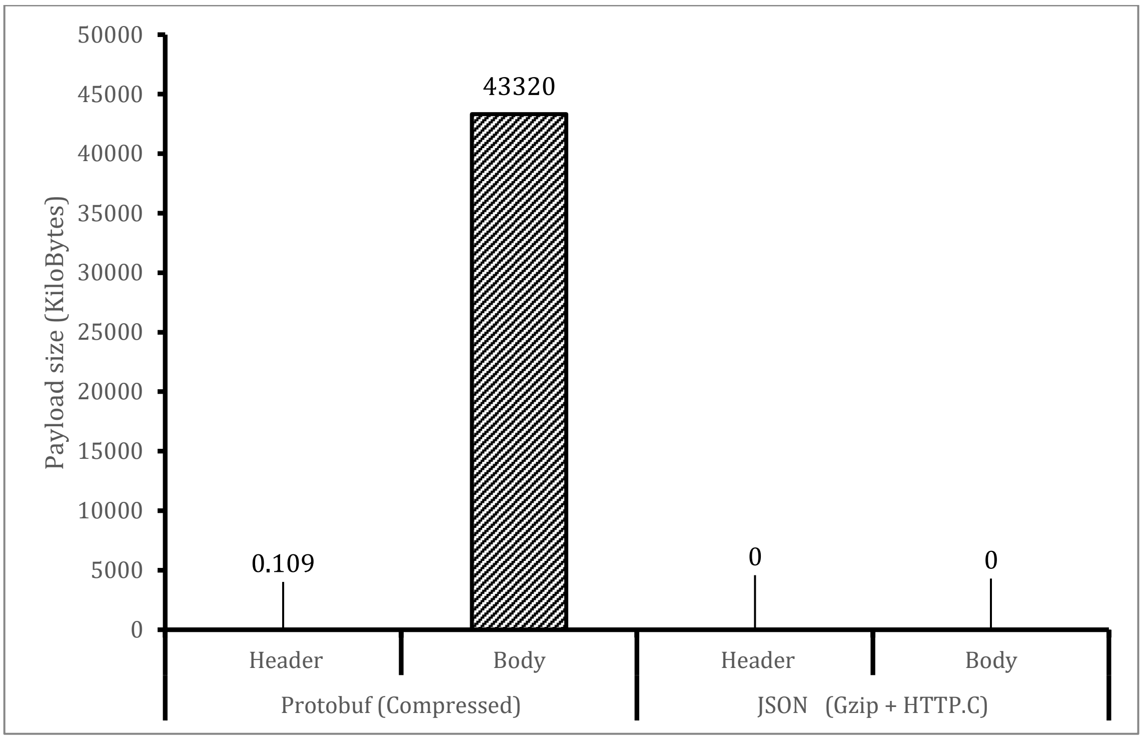


Figure 44 - Payload size benchmark results / Data Generation Count 300 000 / flat data structure / data presented with header and body size

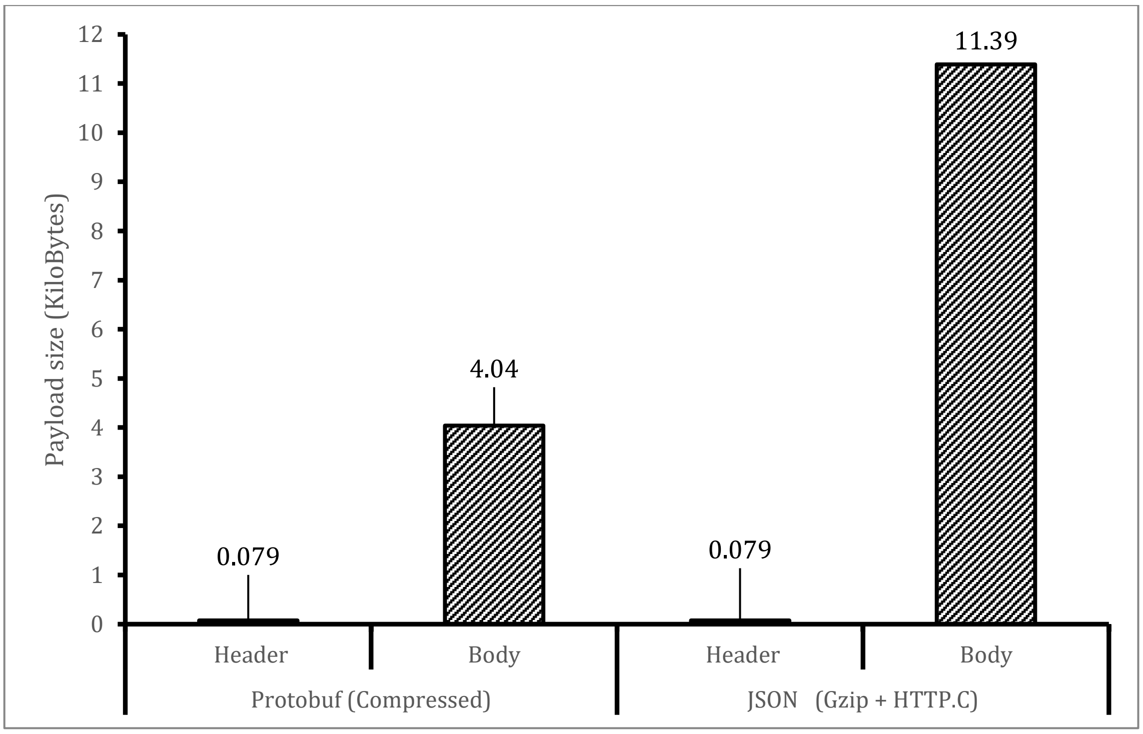


Figure 45 - Payload size benchmark results / Data Generation Count 5 / nested data structure / data presented with header and body size

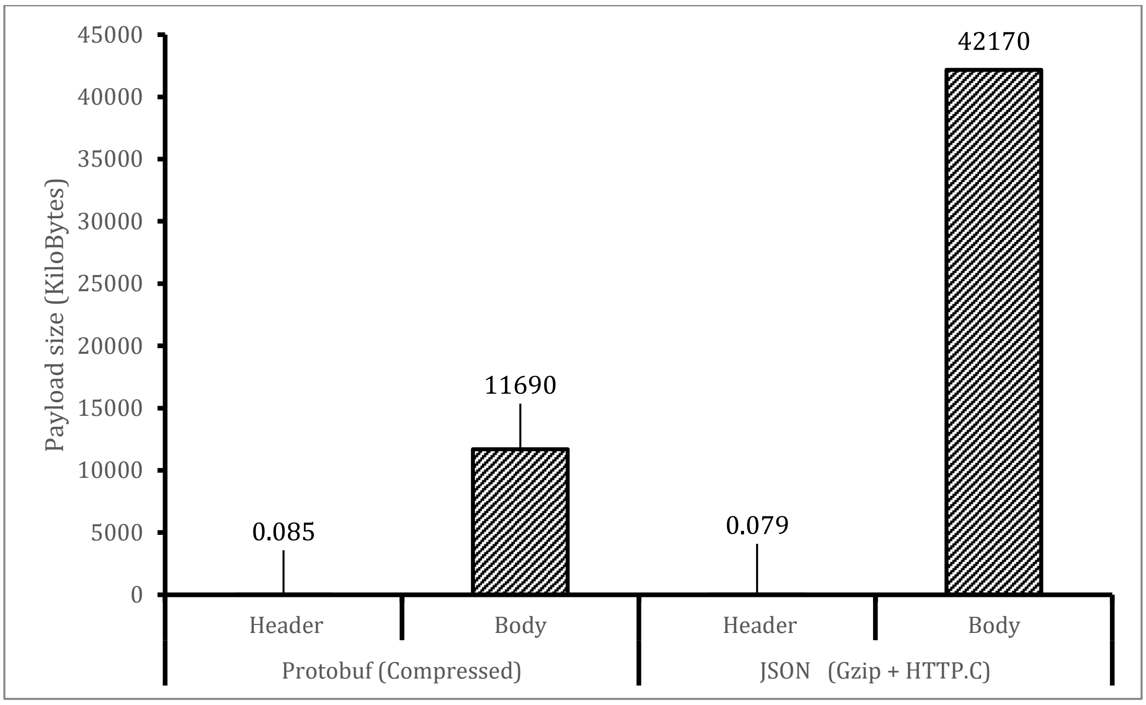


Figure 46 - Payload size benchmark results / Data Generation Count 325 / nested data structure / data presented with header and body size

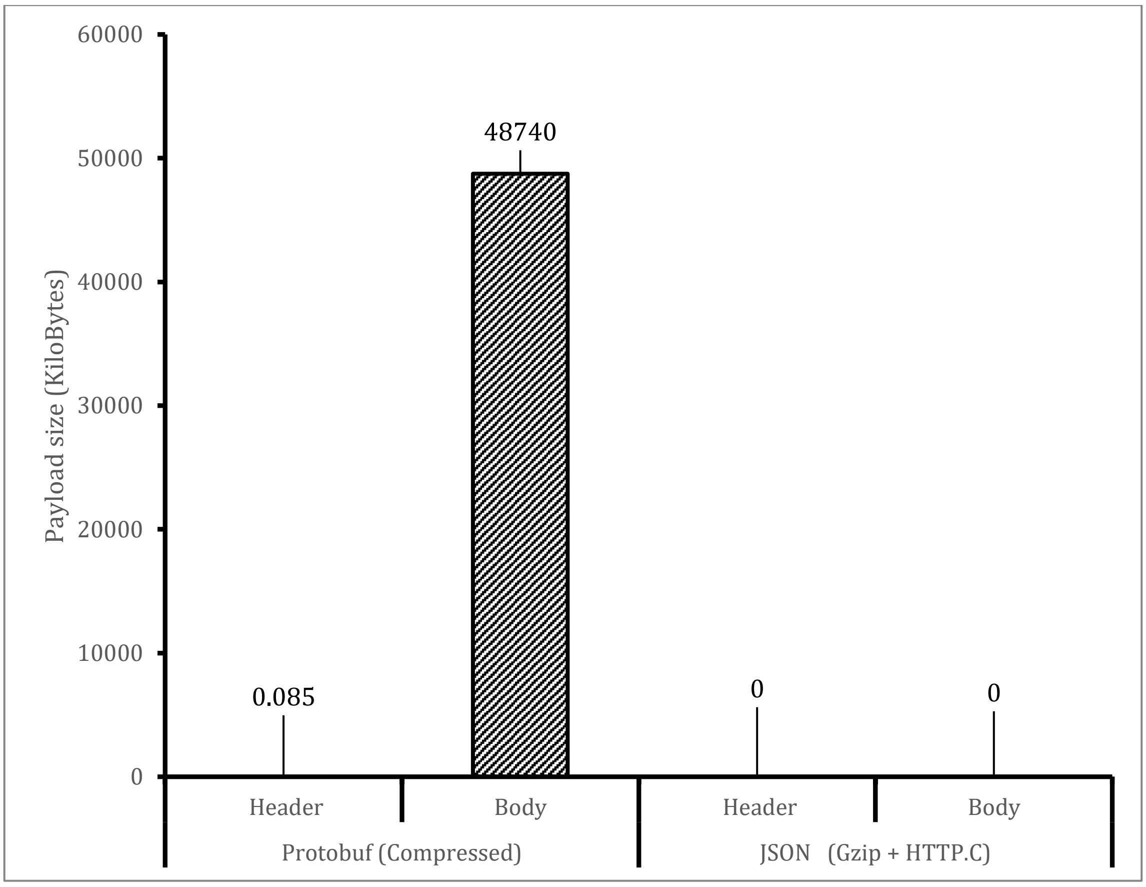


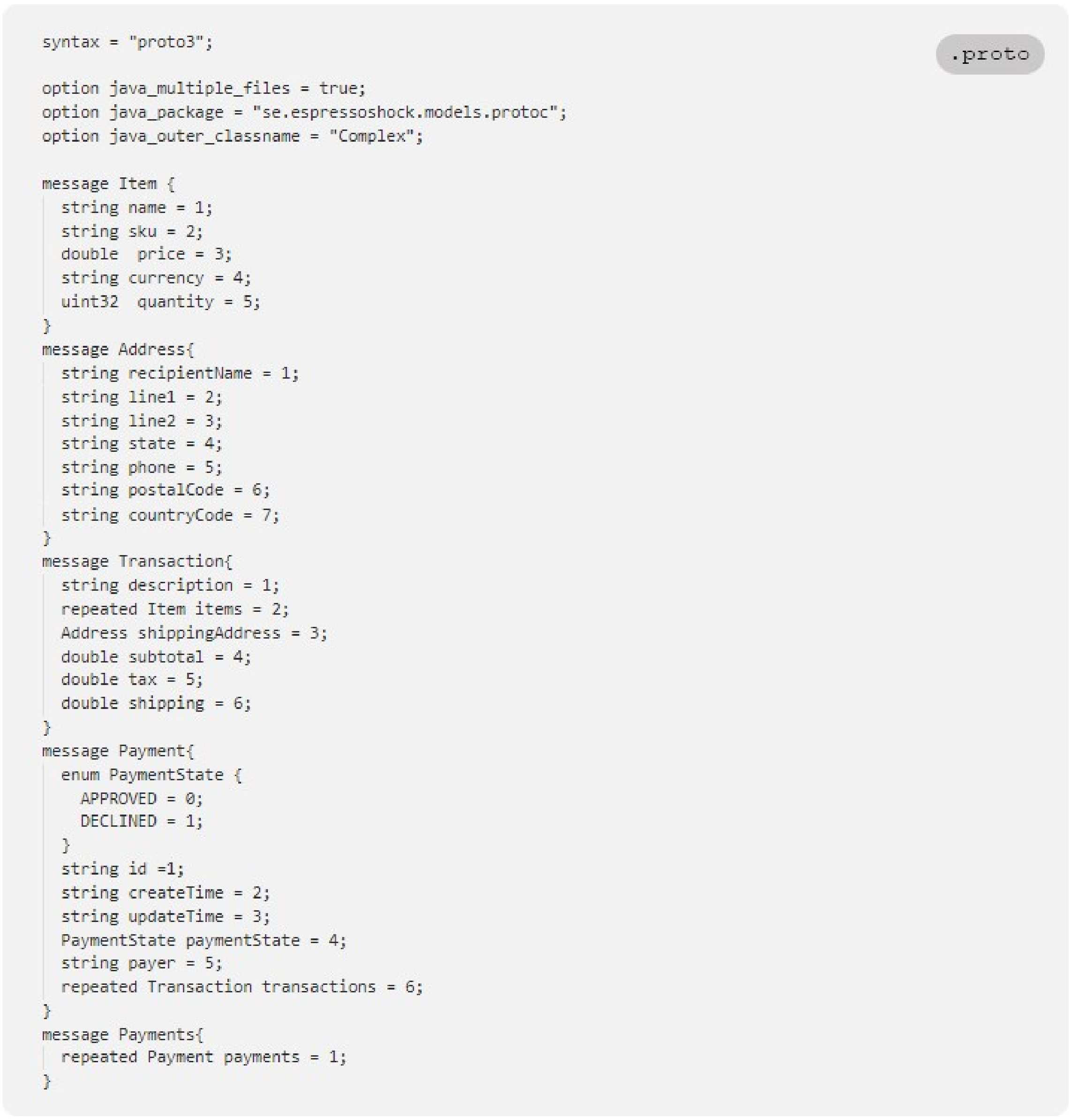
Figure 47 - Payload size benchmark results / Data Generation Count 650 / nested data structure / data presented with header and body size

Appendix 4: Source code

This appendix reports the source code of the following study, including the generate classes compiled by the *Protobuf protoc* compiler. **Figure 48** shows the message type definition of the nested data structure used for the benchmarks.

Source code available at: https://github.com/espressoshock/da399b-supplementary-material-public [72]

Source code of protoc compiled classes: https://github.com/espressoshock/da399b-supplementary-material-public/tree/main/protoc



```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "se.espressoshock.models.protoc";
option java_outer_classname = "Complex";

message Item {
  string name = 1;
  string sku = 2;
  double  price = 3;
  string currency = 4;
  uint32  quantity = 5;
}
message Address{
  string recipientName = 1;
  string line1 = 2;
  string line2 = 3;
  string state = 4;
  string phone = 5;
  string postalCode = 6;
  string countryCode = 7;
}
message Transaction{
  string description = 1;
  repeated Item items = 2;
  Address shippingAddress = 3;
  double subtotal = 4;
  double tax = 5;
  double shipping = 6;
}
message Payment{
  enum PaymentState {
    APPROVED = 0;
    DECLINED = 1;
  }
  string id =1;
  string createTime = 2;
  string updateTime = 3;
  PaymentState paymentState = 4;
  string payer = 5;
  repeated Transaction transactions = 6;
}
message Payments{
  repeated Payment payments = 1;
}
```

Figure 48 - Proto definition of the Payment nested (complex) data structure