

Enhancing Data Serialization Efficiency in REST Services: Migrating from JSON to Protocol Buffers

Anas Shatnawi^a, Adem Bahri, Boubou Thiam Niang^b and Benoit Verhaeghe^c

Berger-Levrault, Mauguio, France

fi

Keywords: REST Services, Data Serialization, JSON, Protocol Buffers, Software Transformation, Migration.

Abstract: Data serialization efficiency is crucial for optimizing web application performance. JSON is widely used due to its compatibility with REST services, but its text-based format often introduces performance limitations. As web applications grow more complex and distributed, the need for more efficient serialization methods becomes evident. Protocol Buffers (Protobuf) has demonstrated significant improvements in reducing payload size and enhancing serialization/deserialization speed compared to JSON. To improve the performance and optimize resource utilization of existing web applications, the JSON data serialization approach of their REST services should be migrated to Protobuf. Existing migration approaches emphasize manual processes, which can be time-consuming and error-prone. In this paper, we propose a semi-automated approach to migrating the data serialization of existing REST services from JSON to Protobuf. Our approach refactors existing REST codebases to use Protobuf. It is evaluated on two web applications. The results show a reduction in payload size by 60% to 80%, leading to an 80% improvement in response time, a 17% decrease in CPU utilization, and an 18% reduction in energy consumption, all with no additional memory overhead.

1 INTRODUCTION


Web applications have become essential in our daily lives. As they exchange data, the efficiency of data serialization (Jang and et al., 2020) plays a critical role in optimizing their performance. Data serialization directly affects resource usage, and system responsiveness (Eugster and et al., 2003). Traditionally, web applications have relied on text-based data serialization approaches, such as JSON (Crockford, 2006) that is integrated with REST services since the early 2000s (Fielding, 2000). These approaches have gained widespread adoption among companies for developing numerous web applications due to their simplicity and ease of use (Barbaglia and et al., 2017).


Text-based data serialization methods often face performance challenges, especially when managing large payloads or complex data structures. The text-based nature of JSON results in larger data sizes being exchanged between application services, which increases both parsing and transmission overhead (Fernando, 2022; Kelly, 2022). As applications be-


come more complex and distributed, the demand for faster and more efficient data serialization approaches grows (Štefanič, 2022; Eugster and et al., 2003), especially for those requiring real-time processing.

As part of their Google Remote Procedure Call (gRPC), Google has introduced advanced version of Protocol Buffers (Protobuf), which provides a compact and efficient binary format for data serialization (Štefanič, 2022). It enhances application performance by minimizing data size and improving serialization/deserialization speeds (Lee and Liu, 2022a; Berg and Mebrahtu Redi, 2023), demonstrating a performance advantage over JSON, particularly with large payloads (Eugster and et al., 2003; Fernando, 2022; Štefanič, 2022; Kelly, 2022). Lee et al. (Lee and Liu, 2022a) show that gRPC can outperform REST by up to 7 and 10 times in data reception and transmission, respectively, due to Protobuf's efficient serialization. Thus, Protobuf presents a robust alternative to JSON for optimizing performance and resource utilization.

To optimize performance and resource utilization in existing web applications, migrating the JSON data serialization of REST services to Protobuf is essential (Štefanič, 2022; Lee and Liu, 2022a; Berg and Mebrahtu Redi, 2023). Companies face two main options: a complete application rebuild or a code

^a  <https://orcid.org/0000-0002-5561-4232>

^b  <https://orcid.org/0000-0002-8618-1740>

^c  <https://orcid.org/0000-0002-4588-2698>

refactoring approach. While a complete rebuild can be resource-intensive and time-consuming, a migration approach offers a more manageable alternative by leveraging the existing codebase. With migration, we need to address two main questions: how to migrate existing REST services from JSON to Protobuf? and what are the benefits of this migration? Existing approaches (Štefanič, 2022; Lee and Liu, 2022a; Berg and Mebrahtu Redi, 2023) emphasize manual processes, which can be time-consuming and error-prone. These approaches often lack performance measurement post-migration, making it challenging to assess their impact on application efficiency. Furthermore, manual migration is typically unsuitable for industrial applications due to its resource-intensive nature, limiting scalability and increasing the risk of errors in larger systems.

In this paper, we propose a semi-automated approach for migrating the data serialization of existing REST services from JSON to Protobuf to enhance their performance. Our approach involves refactoring the codebase of these services to substitute JSON with Protobuf. We evaluate our approach using two web applications—one open-source and one industrial—of varying sizes. The results show a 60% to 80% reduction in payload size, leading to an 80% improvement in response time, a 17% decrease in CPU utilization, and an 18% reduction in energy consumption, all without additional memory overhead.

The rest of this paper is organized as follows. Section 2 presents the challenges and the proposed approach. Section 3 details the identification of Data Transfer Objects (DTOs) and the transformation of their data schema into the Protobuf format. Section 4 explains the generation of Protobuf entities based on the transformed schema. Section 5 describes how existing DTOs are refactored to include Protobuf serialization and deserialization methods. In Section 6, we refactor the REST controllers to replace JSON with Protobuf. Section 7 presents the evaluation results, and related work is analyzed in Section 8. Finally, Section 9 concludes the paper and outlines future research directions.

2 THE PROPOSED APPROACH

2.1 Migration Challenges

Migrating the data serialization of REST services from JSON to Protobuf is not a straightforward task. Unlike the migration from JSON to XML, which in Java and Spring can often be accomplished by simply modifying a Maven dependency or annotations,

this process involves significant refactoring in the application codebase (e.g., source code and configuration files). For instance, all DTOs must be adapted to support the Protobuf format and existing REST controllers need to be updated to use the serialization and deserialization techniques provided by Protobuf.

This task is further complicated by the complexity of DTO classes. Identifying all DTOs across complex applications, especially in large-scale industrial systems, presents several challenges. These include handling nested DTOs, multiple inheritance levels, and DTOs inherited from external libraries. This process demands a detailed analysis of controller methods, inheritance hierarchies, and associated data models. Mapping Java data types, including complex types like List, Set, Map, and enums, to their Protobuf equivalents can be non-trivial. Ensuring precision while handling edge cases (e.g., handling nested or inherited DTOs) is critical. Reverse engineering large and intricate data schemas to prepare Protobuf .proto files demands thorough validation to ensure that all relationships and fields are accurately represented.

Additionally, REST clients must be updated to properly consume the newly Protobuf-based controllers, ensuring compatibility and seamless communication. This involves adapting the client-side logic to serialize and deserialize data in the Protobuf format instead of JSON. Depending on the programming language and framework used by the clients, this may require integrating Protobuf libraries, generating client-side Protobuf classes from the .proto files, and modifying API calls to handle the Protobuf-encoded data.

To facilitate a gradual transition to the new Protobuf-based REST API, it is advisable to maintain both JSON and Protobuf versions of the controllers during the migration phase. This approach enables clients to incrementally adapt to the Protobuf format while continuing to use the existing JSON-based endpoints. By supporting both formats concurrently, developers can ensure backward compatibility, reduce the risk of service disruptions, and allow client-side teams to migrate at their own pace. Over time, as all clients adopt the Protobuf format, the JSON-based controllers can be deprecated and eventually removed, completing the migration process.

2.2 Migration Process

Our approach introduces a semi-automated process to migrate REST API data serialization from JSON to Protobuf. The process, outlined in Figure 1, consists of four main steps: (1) Identify DTOs and reverse engineer the data schema into Protobuf format, (2) Generate Protobuf entities, (3) Refactor DTOs to

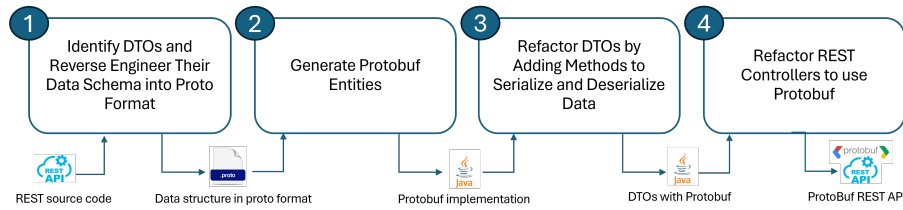


Figure 1: Process for migrating REST from JSON to Protobuf.

include serialization and deserialization methods, and (4) Refactor REST controllers to use Protobuf.

(1) Identify DTOs and Reverse Engineer Their Data Schema into Protobuf Format: This step involves analyzing the data exchanged through REST APIs by identifying the Data Transfer Object (DTO) (Monday, 2003) classes for each REST endpoint. These DTOs define the input parameters and return objects. The data structure of each DTO is reverse-engineered to extract its schema, which is crucial for understanding data organization and transmission between the API and its clients. This schema is then translated into the Protobuf format by creating a `.proto` file.

(2) Generate Protobuf Entities: To facilitate data conversion between Protobuf and Java objects of the business logic, we generate Protobuf entities equivalent to the DTOs. This is based on the data schema recovered in the previous step. These entities handle the data in the REST/Protobuf services produced by our approach and the necessary boilerplate code (Builders) to construct Protobuf instances from DTO instances and vice versa. Using the Protocol Buffers Compiler (*protoc*), the creation of the Protobuf entity classes as well as their inner logic is automated. The compiler reads the `.proto` file and generates the necessary Java classes to manage data structures, ensuring smooth conversion without manually writing complex logic.

(3) Refactor DTOs by Adding Methods to Serialize and Deserialize Data: To avoid modifying the application's business logic, the internal implementation of the REST services continue working with the original DTOs. However, these original DTOs are not capable of converting to or constructing from Protobuf instances that are received or sent by a REST endpoint. Thus, this step aims to refactor existing DTO classes by adding methods for serialization and deserialization of data using the generated Protobuf classes. This integration allows DTOs to efficiently convert data to and from Protobuf instances.

(4) Refactor REST Controllers to use Protobuf: This step modifies existing REST controllers to handle Protobuf instead of JSON. Rather than completely replacing the existing controller, a new Protobuf-

based controller is generated. This dual-controller approach allows both JSON and Protobuf versions to co-exist during the migration, enabling a gradual transition without disrupting current REST clients.

3 IDENTIFY DTOS AND TRANSFORM THEIR DATA SCHEMA INTO PROTOBUF

In this section, we discuss how to identify the DTOs within our REST services and convert their data schema into the Protobuf format. To do so, we first analyze the data models associated with all REST endpoints across the REST controllers. This involves scanning each controller's methods to examine both the input parameters, which represent the data received from the client, and the return types, which represent the data returned to the client.

Once the DTOs are identified, we reverse engineer their data schema to prepare for migration to Protobuf. We examine the fields of each DTO to identify their Java data types. We also check the inheritance of each DTO, as their parent classes must be migrated to Protobuf as well.

To establish a mapping between Java types and their corresponding Protobuf types, we rely on the Protobuf documentation (protobuf.dev, 2024). For instance, Java's primitive types such as *int*, *long*, and *boolean* directly translate to Protobuf types *int32*, *int64*, and *bool*, respectively. Complex types like *List* and *Set* in Java are represented as repeated fields in Protobuf, while *Map* is directly mapped using *map<KeyType, ValueType>*. Additionally, other types, including *enum*, *double*, *float*, and *String*, each have their specific equivalents in Protobuf.

Based on this mapping, we derive the Protobuf format from these data schemas by creating a `.proto` file. To do so, we propose an algorithm illustrated in Figure 2. It begins by creating a new `.proto` file and adding the necessary header information, including package details, import statements, and the syntax declaration (e.g., `syntax = "proto3"`; specifies the version of Protobuf). Then, it parses a given Java

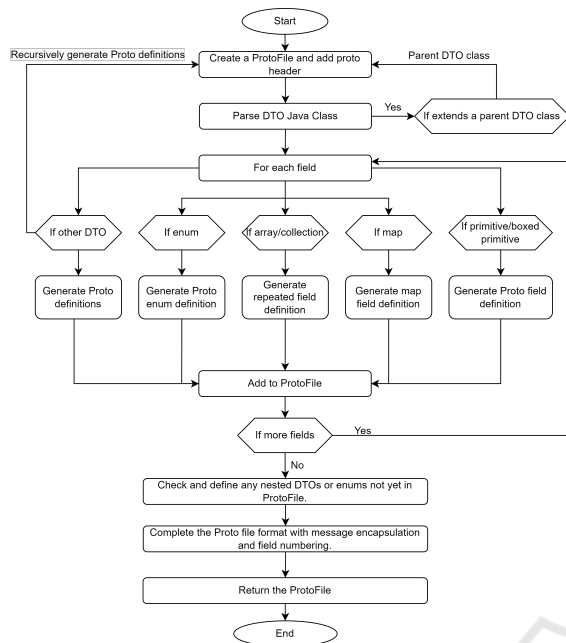


Figure 2: Transformation from DTOs to Protobuf format.

DTO class to extract its fields and their data types. If the DTO extends a parent class, it recursively processes the parent DTO before continuing with the current one. As the algorithm iterates over each field, it determines the field type and generates the appropriate Protobuf definition. If the field is another DTO, the algorithm recursively generates Protobuf definitions for the nested DTO. For enums, a corresponding Protobuf *enum* definition is created. Arrays and collections are transformed into repeated fields, while maps are generated as *map<KeyType, ValueType>*. For primitive or boxed primitive types, the algorithm directly generates a Protobuf field definition (e.g., Long to *sint64*). Each generated field is then added to the *.proto* file. Throughout the process, the algorithm ensures that all referenced nested DTOs or enums are properly defined in the ProtoFile. After processing all fields and confirming message encapsulation, the *.proto* file is finalized and returned.

4 GENERATE PROTOBUF ENTITIES

In this step, we generate Protobuf entities based on the data schema embedded in the *.proto* file generated in the previous step. To generate these Protobuf entities from *.proto*, we rely on the Protobuf compiler (*protoc*) (gRPC, 2024) provided by Google to generate Protobuf entities based on a given *.proto* file. This based on three main steps: (1) **Add the *proto***

Plugin: we ensure that the *protoc* compiler plugin is added to the build configuration of the target project. For a Maven project, we need to include the *protobuf-maven-plugin*¹ in the *pom.xml* file. (2) **Run the *protoc* Compiler:** we execute the build command to compile the *.proto* file and generate the Java classes. For Maven, use: *mvn clean install*. This command triggers the build process, which will compile the *.proto* files and produce the corresponding Java classes in the *target/generated-sources/protobuf* directory. (3) **Verify the Output:** After running the build, we ensure that the Java files have been generated. These files include classes for implementing the Protobuf messages (corresponding to DTOs) defined in the *.proto* file.

This process integrates the *protoc* compiler into the build workflow, automating the generation of Java classes from Protobuf definitions and ensuring that the application can effectively handle Protobuf data.

5 REFACTOR DTOS BY ADDING METHODS TO SERIALIZE AND DESERIALIZE DATA

To refactor the existing DTOs by adding serialization and deserialization methods in Java, we utilize the Protobuf entities generated in the previous step. These methods are responsible for converting Java objects used in the business logic into Protobuf messages and vice versa. For each DTO, we add *toProto* and *fromProto* methods, which respectively convert the DTO object to a Protobuf message and the Protobuf message back to a DTO object.

To implement the *toProto* method, we utilize the generated Protobuf entities, which include *Builder* objects for constructing Protobuf messages. We use the setter methods of the *Builder* to fill the Protobuf message with values from the corresponding DTO fields (e.g., *builder.setId(id)*). For *fromProto* method, we extract each field from the received Protobuf instance using its getter methods. With the extracted values, the method creates a new instance of the DTO class, ensuring that all data from the Protobuf message is accurately transferred to the Java object.

To automate the generation of the *toProto()* and *fromProto()* methods, we propose an algorithm outlined in Figure 3. It begins by taking a Protocol Buffers (*.proto*) file as input and using the Protoc compiler to generate the initial Java classes for the corresponding Protobuf message types. Once these classes are generated, an Abstract Syntax Tree (AST)

¹<https://github.com/xolstice/protobuf-maven-plugin>

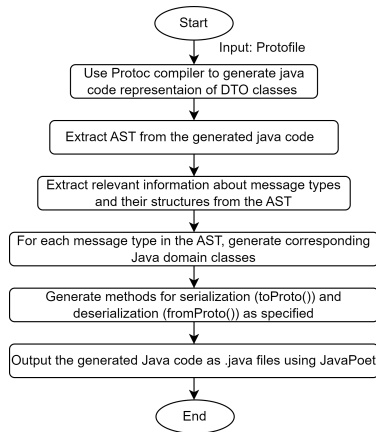


Figure 3: Refactor DTOs by adding methods to serialize and deserialize data.

is extracted from the Java code. The AST provides a structured overview of the class hierarchies, fields, methods, and data types. We then analyze this structure to identify the relevant Protobuf message types and map them to their respective fields within the DTO classes. For each Protobuf message type, we generate a corresponding Java class, which includes fields mapped directly from the *.proto* file. To achieve this, we leverage the JavaPoet library², which programmatically generates the necessary Java code, producing the final *.java* files. This automated approach ensures consistency, reduces manual effort, and minimizes the potential for errors in the serialization and deserialization process. We implement this algorithm in called *ProtoGen*³ that streamlines the integration of Protobuf with existing Java applications.

6 REFACTOR JSON CONTROLLER TO PROTOBUF CONTROLLER

This step modifies existing REST controllers to handle Protobuf messages while keeping the original JSON-based controllers intact. A new Protobuf-based controller is generated, allowing both JSON and Protobuf versions to coexist. This dual-controller approach enables a gradual migration, ensuring backward compatibility without disrupting current REST clients. Controller methods are refactored by updating return types and parameters to use Protobuf messages, replacing JSON-based data structures. The *toProto()* and *fromProto()* methods ensure efficient handling of Protobuf requests and responses.

²<https://github.com/palantir/javapoet>

³<https://github.com/Bahri-Adem/ProtoGen>

We propose an algorithm illustrated in Figure 4. It begins by retrieving all Java files in the project directory. Each file is processed individually to determine if it contains classes annotated with REST controller annotations, such as `@RestController` or `@RequestMapping`. Once a REST controller is identified, we create a copy of the class and place it in a new package designated for Protobuf REST services. The package declaration in the copied class is updated to reflect its new location. Necessary Protobuf entities and classes are imported into the copied file. Next, the class definition is modified by updating its annotation and class name to indicate that it is now a Protobuf controller. For each endpoint in this controller, we refactor its Java method as follows. We adjust the return type to correspond to the appropriate Protobuf message type. Then, the method parameters are refactored to match the Protobuf request message format. Additionally, the return expressions are modified to incorporate Protobuf-specific logic for handling requests and responses. This process is repeated for any other REST controllers present in the project. It's worth noting that we do not need to modify the business logic of these methods.

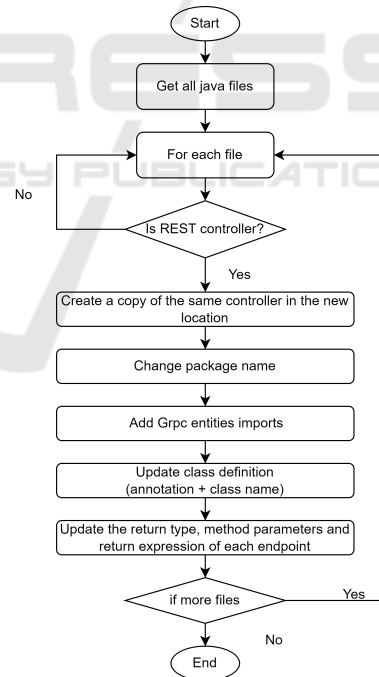


Figure 4: Controller converter algorithm.

7 EVALUATION

7.1 Evaluation Goal and Methodology

Our approach is evaluated by assessing the efficiency of Protobuf compared to JSON in terms of application performance. This analysis enables us to quantify the benefits of adopting Protobuf as a data serialization approach. To do so, we rely on two web applications. Application 1 is the PetStore, an open-source application proposed by Oracle to demonstrate Java technologies. Application 2 is a large-scale industrial application from the Berger-Levrault company.

To assess the impact of serialization formats, we created two controllers for the same application: one with REST/JSON and the other with REST/Protobuf. To collect metrics, we began by identifying identical workloads to ensure fair comparisons, using consistent real-world scenarios to ensure that performance differences reflect the serialization formats used. Then, we executed large workload sizes under varied conditions, including concurrent users and varying data volumes, while functional tests helped determine appropriate workload sizes for comprehensive data generation. Finally, we continuously collect performance metrics during workload execution.

Our evaluation is based on five key metrics: (1) **Data exchange size**: The size of data transferred between clients and servers. It indicates the cost of parsing and transmitting this data. (2) **Response time**: The duration from request to response. It is connected to the user experience. (3) **CPU usage**: The percentage of processing power consumed during request handling. It is used to evaluate the system's efficiency under load. (4) **Energy consumption**: The power used during request processing. It is measured to assess its sustainability. (5) **Memory usage**: The amount of memory consumed during request execution. It is related to resource utilization.

7.2 Results

Figure 5 illustrates the percentage reduction achieved with Protobuf compared to JSON in terms of data exchange size, response time, CPU usage, energy consumption, and memory usage. The results show that Protobuf achieves substantial savings, with Application 1 experiencing up to an 80% reduction at 16,384 objects (from 1.69 MB to 0.36 MB), and Application 2 seeing a 60% reduction at 256 objects (from 2.23 MB to 0.84 MB). This reduction is largely due to Protobuf's more compact binary format, which is optimized for data transmission, unlike JSON's verbose text-based format. These results indicate that

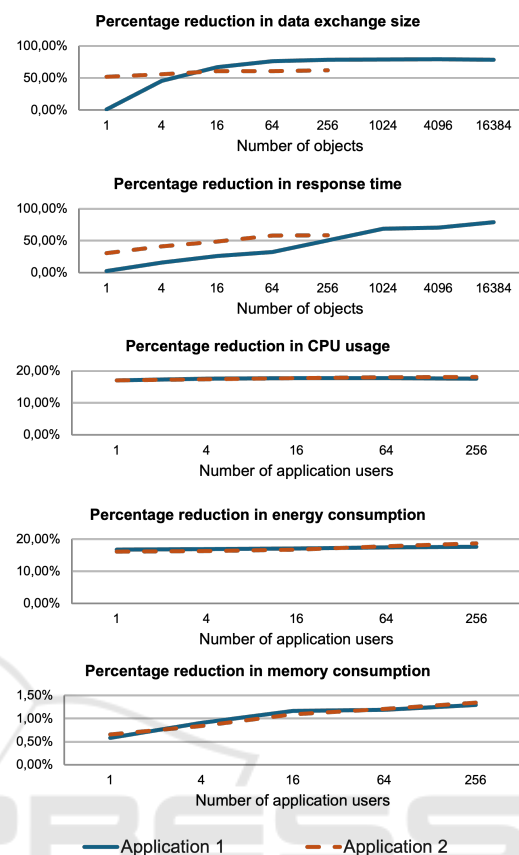


Figure 5: Results of percentage reduction achieved with Protobuf compared to JSON.

Protobuf is especially efficient as the number of objects increases, reducing transmission overhead and network latency. The consistent reduction in data size across both applications suggests that Protobuf can be highly beneficial for systems handling large datasets. Based on these findings, adopting Protobuf in data-intensive applications is recommended, as it can significantly improve performance by minimizing data payload size. For response time, Application 1 exhibits around an 80% reduction (from 2969 MS to 631 MS), while Application 2 shows approximately a 60% reduction (from 3657 MS to 2141 MS). These results are due to Protobuf's more efficient serialization and deserialization processes compared to JSON, which involves parsing more complex text structures. The results indicate that Protobuf significantly improves response times, particularly as the data load increases, making it a better choice for high-performance applications. This performance boost suggests that systems requiring fast, real-time responses can benefit greatly from switching to Protobuf. Based on these findings, adopting Protobuf in latency-sensitive applications is highly recommended, especially where large data transfers or frequent interactions are in-

volved. Concerning the CPU usage, the results show that Protobuf reduces CPU usage by about 17% in Application 1 and 18% in Application 2. This reduction can be attributed to Protobuf which requires significantly less processing power for serialization and deserialization compared to JSON. As a result, Protobuf's efficiency becomes particularly evident under high user loads, where it helps reduce CPU strain and enhances overall system scalability. These findings suggest that Protobuf can lead to more efficient resource utilization, especially in environments where CPU resources are limited. For energy consumption, the results illustrate that reduction is around 18% and 19% for Application 1 and Application 2, respectively. This significant energy saving is attributed to Protobuf's efficiency in data serialization and deserialization processes, which require fewer resources for parsing and transmitting smaller data. Regarding memory consumption, the results show that the maximum memory reduction observed was around 1.3% for Application 1 and 1.4% for Application 2 under heavy user loads. However, this change is negligible, as we do not have any significant impact on memory usage. This is a good indication that the migration to Protobuf yields positive results without introducing any negative effects on memory consumption.

In summary, Protobuf consistently outperforms JSON across all tested performance metrics, demonstrating advantages in data size, response time, CPU usage, and energy consumption, with no additional memory overhead. This positions Protobuf as a compelling choice for applications requiring efficient serialization and deserialization, especially in high-performance and resource-constrained environments.

7.3 Threats to Validity

Internal Threats. While a controlled testing environment is useful for isolating variables, it might not reflect real-world complexities like network latency or fluctuating system load. Also, simulated workloads, although designed to mimic typical user interactions, may not capture the intricacies of live environments, which could lead to discrepancies in performance metrics.

External Threats. While Protobuf is supported by various programming languages, the results obtained in this study specifically represent its performance with Java web applications. However, Java-based backend applications represent a wide range of industrial applications. To generalize these findings for other languages (e.g., JavaScript), an empirical study should be conducted.

8 RELATED WORK

Several approaches have been proposed to enhance the performance of web applications by migrating to more efficient technologies (Kennedy and Molloy, 2009; Verhaeghe and et al., 2021; Darbord and et al., 2023; Kaushalya and Perera, 2021; Rabetski and Schneider, 2013; Lee and Liu, 2022a). While our approach focuses on migrating the data serialization, they address other aspects of web application migration. Some approaches emphasize communication protocol migration, including migration from RPC to REST (Kennedy and Molloy, 2009), REST to gRPC (Lee and Liu, 2022b), as well as RMI to REST (Darbord and et al., 2023). Other ones migrate the frontend framework including GWT to Angular (Verhaeghe and et al., 2021), AngularJS to Web Components (Ronde, 2021), Thymeleaf to Angular (Eriksson, 2022), and AngularJS to React (Kaushalya and Perera, 2021). On the architectural side, attention has been given to migrating from monolithic architectures to microservice-based ones. Pinos et al. (Pinos-Figueroa and León-Paredes, 2023) propose a refactoring strategy, while others recommend layered architecture designs (Zaragoza and et al., 2022) or automated graph clustering techniques to decompose monolithic systems using data flow-driven approaches (Filippone and et al., 2023; Chen et al., 2017). Other efforts focus on migration to cloud-native architectures (Balalaie and et al., 2016), or shifting from on-premises systems to Software as a Service (SaaS) (Rabetski and Schneider, 2013; Sabiri and Benabbou, 2015). Although these approaches contribute to improving web applications, they do not address the migration of data serialization formats.

Few approaches specifically target migrating data serialization from JSON to Protobuf. Most focus on transitioning from REST to gRPC but overlook a thorough Protobuf migration. Yunhyeok Lee and Yi Liu (Lee and Liu, 2022a) propose a manual approach for migrating REST to gRPC using a simple case study, but it lacks scalability for larger applications and does not include performance evaluation. Michal Štefanič (Štefanič, 2022) offers guidelines for migrating from REST microservices to gRPC but overlooks the migration from JSON to Protobuf. Although the study uses an industrial example, it lacks automation techniques for streamlining the migration process. Additionally, while the study discusses data exchange size, it does not measure key performance aspects like response time, memory consumption, or energy efficiency. Berg and Redi (Berg and Mebrahtu Redi, 2023) focus on analyzing data exchange size and latency but omit key performance metrics

such as CPU usage, memory consumption, and energy efficiency, which are crucial for a comprehensive evaluation. Additionally, their testing relies on a small example and lacks migration guidelines.

9 CONCLUSION

We propose a semi-automated approach for migrating the data serialization of existing REST services from JSON to Protobuf. This aims to optimize the performance of existing web applications. We evaluate our approach using one industrial and one open-source applications. The results show that both applications achieve similar levels of performance optimization and resource utilization reduction. They exhibit a reduction in data size of 80% for larger payloads, improve response times by 80%, reduce CPU utilization by 17%, and decrease energy consumption by 18%, all without consuming any additional memory. Protobuf emerges as a strong contender for organizations looking to optimize their applications' performance and responsiveness. Therefore, it is highly recommended to consider Protobuf in scenarios characterized by high user concurrency and stringent resource constraints, as it not only improves performance but also contributes to more sustainable software.

As a future direction, we plan to extend our approach by migrating the communication protocol from REST to gRPC and testing it with a wider range of web applications. We plan to explore the migration to Protobuf in frontend applications.

REFERENCES

- Balalaie, A. and et al. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33(3):42–52.
- Barbaglia, G. and et al. (2017). Definition of rest web services with json schema. *Software: Practice and Experience*, 47(6):907–920.
- Berg, J. and Mebrahtu Redi, D. (2023). Benchmarking the request throughput of conventional api calls and grpc: A comparative study of rest and grpc.
- Chen, R., Li, S., and Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475. IEEE.
- Crockford, D. (2006). The application/json media type for javascript object notation (json). *Internet Engineering Task Force IETF Request for Comments*.
- Darbord, G. and et al. (2023). Migrating the communication protocol of client-server applications. *IEEE Software*, 40(4):11–18.
- Eriksson, J. (2022). Migration of the user interface of a web application: from thymeleaf to angular.
- Eugster, P. T. and et al. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131.
- Fernando, R. (2022). Evaluating performance of rest vs. grpc. <https://medium.com/@EmperorRXXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da>.
- Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.
- Filippone, G. and et al. (2023). From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In *ICSA*, pages 47–57. IEEE.
- gRPC (2024). <https://grpc.io/docs/protoc-installation/>. (accessed: September 2024).
- Jang, J. and et al. (2020). A specialized architecture for object serialization with applications to big data analytics. In *ISCA*, pages 322–334. IEEE.
- Kaushalya, T. and Perera, I. (2021). Framework to migrate angularjs based legacy web application to react component architecture. In *Moratuwa Engineering Research Conference (MERCon)*, pages 693–698. IEEE.
- Kelly, T. (2022). grpc vs rest: Performance benchmarking. *Journal of Systems Research (JSys)*.
- Kennedy, S. and Molloy, O. (2009). A framework for transitioning enterprise web services from xml-rpc to rest.
- Lee, Y. and Liu, Y. (2022a). Using refactoring to migrate rest applications to grpc. *University of Massachusetts Dartmouth*.
- Lee, Y. and Liu, Y. (2022b). Using refactoring to migrate rest applications to grpc. In *ACM Southeast Conference*, pages 219–223.
- Monday, P. (2003). Implementing the data transfer object pattern. In *Web Services Patterns: Java™ Platform Edition*, pages 279–295. Springer.
- Pinos-Figueroa, B. and León-Paredes, G. (2023). An approach of a migration process from a legacy web management system with a monolithic architecture to a modern microservices-based architecture of a tourism services company. In *CONISOFT*, pages 9–17. IEEE.
- protobuf.dev (2024). <https://protobuf.dev/programming-guides/proto3/#scalar>. (accessed: Sep/2024)
- Rabetski, P. and Schneider, G. (2013). Migration of an on-premise application to the cloud: Experience report. In *ESOCC*, pages 227–241. Springer.
- Ronde, S. (2021). *Migrating Angular-based web apps to Web Components-A case study at 30MHz*. PhD thesis, Universiteit van Amsterdam.
- Sabiri, K. and Benabbou, F. (2015). Methods migration from on-premise to cloud. *Journal of Computer Engineering*, 17(2):58–65.
- Verhaeghe, B. and et al. (2021). Migrating gui behavior: from gwt to angular. In *ICSME*, pages 495–504. IEEE.
- Zaragoza, P. and et al. (2022). Leveraging the layered architecture for microservice recovery. In *ICSA*, pages 135–145. IEEE.
- Štefanič, M. (2022). Developing the guidelines for migration from restful microservices to grpc. *Masaryk University*.