INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Migrar aplicações REST para gRPC

HUGO MIGUEL MENDES AMARAL
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Migrating REST applications to gRPC

Hugo Amaral

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Advisor: Dr. Isabel Azevedo

Porto, September 28, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

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

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

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

ISEP, Porto, September 28, 2025

Hugo Amaral

Abstract

The communication between services in distributed architectures in microservices has traditionally been supported by REST, due to its simplicity, generalized adoption, and easy integration with multiple platforms. However, as the systems grow in scale and complexity, relevant limitations associated with REST emerge. It is hard to impose, it is built on top of HTTP 1.X and makes use of text formats readable by humans, which is inefficient for service-to-service interactions and there's the absence of well-defined and strongly typed service definitions. Those limitations become even clearer on systems that require high performance, low latency and strong consistency in service communication.

Google's gRPC presents an as alternative capable of filling those gaps. In particular, gRPC outperforms the traditional REST paradigm, particularly in inter-service communication within microservices architectures, in key metrics such as throughput, response time, bandwidth efficiency, and bi-directional streaming. Moreover, as gRPC uses Protocol Buffers to define services, gRPC service contracts clearly define the types that will be used for interaction between the applications. This helps in overcoming common runtime and interoperability errors that are typically faced when applications are built by multiple teams and different technologies.

However, the adoption of gRPC in already developed REST systems is not a straightforward process. It involves technical and organizational challenges, which go from ensuring the compatibility of clients and servers during the transition to the lack of good practices. In addition, the information about how to effectively migrate REST applications to gRPC is extremely limited, as there is a significant gap in academic literature regarding a methodical and strategic way to perform a migration.

The work's objective, at a high level, is to explore, design and compare migration approaches of REST projects to gRPC, and implement one or more of the developed strategies, providing an extensive analysis of the results. That is achieved by delving particularly into gRPC, but also studying other frameworks, dissecting their differences and identifying their benefits and downsides and what led to their adoption. In the end, the final goal is to provide helpful insights and guidelines for engineering teams studying modernization in their communication protocols, contributing to the broader discussion about the transformation and progression of distributed systems and API architectures.

For that, a systematic literature review was conducted and then complemented by an analysis of real cases in the industry, in order to identify methodologies, challenges, tools and good practices relevant for the migration process. The systematic review revealed a lack of scientific sources directly focused on a migration from REST to gRPC, with the contributions from technical blogs from companies like WePay, Google Cloud and LinkedIn being very helpful. Those cases showed that a migration can be successful with a proper plan, adoption of gradual strategies, automation tools, and thorough testing.

The adopted methodology is based on the practical application of the obtained knowledge by migrating an open-source project, which represents modern microservice-based architectures. The process began with the detailed analysis of each service, endpoint mapping and understanding the interaction between components. Then, the gRPC contracts were defined following strict structure and naming conventions and centralized in a separate repository for the effect in order to make the management, versioning, and integration easier. The automatic generation of code to servers, clients and gateway was done using Protocol Buffer Compiler and the necessary plugins for Java and Go.

During the migration, special attention was given to the coexistence of REST and gRPC, resorting to the gRPC Gateway to ensure that REST clients kept working without any changes. This approach allowed a gradual and smooth transition, while minimizing the risks. The validation of the migration's success was made with automatic and manual tests, especially to ensure functional equivalence after the process is complete. The experience also demonstrated the importance of automation mechanisms in CI pipelines for the generation, publishing and validation of stubs.

Among the faced challenges, the lack of a formal API Specification for REST APIs, which requires a manual analysis of the code, and the low test coverage are highlighted. Nevertheless, the acquired experience allowed the definition of a clear set of guidelines to support engineering teams planning a transition.

The results showed that the migration of REST systems to gRPC is viable and beneficial, as long as it is thoroughly planned, has a good level of automation, and good engineering practices. The adoption of bridge tools and contract centralization allows for minimizing the risks and ensuring the operational uninterruptedness of the systems during the migration. The experience also highlights the need for as much documentation, testing, and automation as possible.

Keywords: gRPC, REST, software migration

Resumo

A comunicação entre serviços em arquiteturas distribuídas de microserviços tem sido tradicionalmente desenvolvida em REST, devido à sua simplicidade, adoção generalizada e fácil integração com múltiplas plataformas. No entanto, à medida que os sistemas crescem em escala e complexidade, surgem limitações relevantes associadas ao REST. É difícil de impor, é desenvolvido em HTTP 1.X e faz uso de formatos de texto legíveis por humanos, o que é ineficiente para interações entre serviços, e há a ausência de definições de serviço bem definidas e fortemente tipadas. Essas limitações tornam-se ainda mais claras em sistemas que requerem alta performance, baixa latência e forte consistência na comunicação entre serviços.

O gRPC, desenvolvido pela Google, apresenta-se como uma alternativa capaz de preencher essas lacunas. Em particular, o gRPC ultrapassa o paradigma REST tradicional, especialmente na comunicação inter-serviços dentro de arquiteturas de microserviços, em métricas importantes como throughput, tempo de resposta, eficiência de largura de banda e streaming bidirecional. Além disso, como o gRPC usa Protocol Buffers para definir serviços, os contratos de serviço gRPC definem claramente os tipos que serão usados para interação entre as aplicações. Isso ajuda a ultrapassar erros comuns de runtime e interoperabilidade que são tipicamente enfrentados quando aplicações são construídas por múltiplas equipas e tecnologias diferentes.

No entanto, a adoção do gRPC em sistemas REST já desenvolvidos não é um processo direto. Envolve desafios técnicos e organizacionais, que vão desde garantir a compatibilidade de clientes e servidores durante a transição até a falta de boas práticas. Além disso, a informação sobre como migrar efetivamente aplicações REST para gRPC é extremamente limitada, pois há falta de literatura académica no que se refere a uma forma metódica e estratégica de realizar uma migração.

O objetivo do trabalho, num nível mais abrangente, é explorar, desenhar e comparar abordagens de migração de projetos REST para gRPC, e implementar uma ou mais das estratégias desenvolvidas, fornecendo uma análise extensa dos resultados. Isso é alcançado aprofundando particularmente o gRPC, mas também estudando outras frameworks, dissecando as suas diferenças e identificando os seus benefícios e desvantagens e o que levou à sua adoção. No final, o objetivo final é fornecer diretrizes para equipas de engenharia que estão a estudar uma modernização nos seus protocolos de comunicação, contribuindo para a discussão mais ampla sobre a transformação e progressão de sistemas distribuídos e arquiteturas de API.

Para isso, foi conduzida uma revisão sistemática da literatura e depois complementada por uma análise de casos reais na indústria, de forma a identificar metodologias, desafios, ferramentas e boas práticas relevantes para o processo de migração. A revisão sistemática revelou uma falta de fontes científicas diretamente focadas numa migração de REST para gRPC, sendo as contribuições de blogs técnicos de empresas como WePay, Google Cloud e LinkedIn muito úteis. Esses casos mostraram que uma migração pode ser bem-sucedida

com um plano adequado, adoção de estratégias graduais, ferramentas de automação e testes rigorosos.

A metodologia adotada baseia-se na aplicação prática do conhecimento obtido através da migração de um projeto open-source, que representa arquiteturas modernas baseadas em microserviços. O processo começou com a análise detalhada de cada serviço, mapeamento de endpoints e compreensão da interação entre componentes. Em seguida, os contratos gRPC foram definidos seguindo uma estrutura rigorosa e convenções de nomenclatura e centralizados num repositório separado para o efeito, de forma a tornar a gestão, versionamento e integração mais fáceis. A geração automática de código para servidores, clientes e gateway foi feita usando o Protocol Buffer Compiler e os plugins necessários para Java e Go.

Durante a migração, foi dada especial atenção à coexistência de REST e gRPC, recorrendo ao gRPC Gateway para garantir que os clientes REST continuassem a funcionar sem quaisquer alterações. Esta abordagem permitiu uma transição gradual e suave, minimizando os riscos. A validação do sucesso da migração foi feita com testes automáticos e manuais, especialmente para garantir a equivalência funcional após o processo estar completo. A experiência também demonstrou a importância de mecanismos de automação em pipelines CI para a geração, publicação e validação de stubs.

Entre os desafios enfrentados, destacam-se a falta de uma especificação de API formal para APIs REST, que requer uma análise manual do código, e a baixa cobertura de testes. Não obstante, a experiência adquirida permitiu a definição de um conjunto claro de diretrizes para apoiar equipas de engenharia que planeiam uma transição.

Os resultados mostraram que a migração de sistemas REST para gRPC é viável e benéfica, desde que seja minuciosamente planeada, tenha um bom nível de automação e boas práticas de engenharia. A adoção de ferramentas de que fazema a ponte entre sistemas e centralização de contratos permite minimizar os riscos e garantir a continuidade operacional dos sistemas durante a migração. A experiência também destaca a necessidade de tanta documentação, testes e automação quanto possível.

Acknowledgement

In first place, I would like to express my deepest gratitude to my family for their unconditional support, patience and incentive during the whole journey. Without their support, this work would not have been possible.

A special thanks to my supervisor, Professor Isabel Azevedo, who tirelessly gave valuable suggestions, which were essential to the development of the dissertation.

Finally, I would like to thank all my colleagues who somehow positively contributed to this journey. Especially João Campelo, Bruno Ferreira, and Francisco Silva, who provided amazing discussions and constant encouragement, making this journey much more enjoyable.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AI	Artificial Intelligence.
API	Application Programming Interface.
CASP	Critical Appraisal Skills Programme.
CI	Continuous Integration.
DTO	Data Transfer Object.
gRPC	Google Remote Procedure Call.
GUI	Graphical User Interface.
HTTP	Hyper Text Transfer Protocol.
IDL	Interface Definition Language.
JAR	Java Archive.
JSON	Javascript Object Notation.
PDL	Pegasus Definition Language.
PICOC	Population, Intervention, Comparison, Outcome, Context.
REST	Representational State Transfer.
RPC	Remote Procedure Call.
SOAP	Simple Object Access Protocol.
XML	eXtensible Markup Language.

Chapter 1

Introduction

In the introduction chapter, the overall objective and methodology of the work is described. After giving an explanation of the context of the problem and the problem itself, it is described the research method.

1.1 Context

Roy Fielding, in his doctoral dissertation (Fielding 2000), introduced Representational State Transfer (REST): a set of constraints that emphasize stateless, resource-oriented architecture, providing a light way for service communication. This contribution altered the trajectory of web architecture and the development of web Application Programming Interface (API), becoming a key paradigm for the creation of such interfaces even in contemporary times (POSTMAN 2023).

Before REST became popular, Simple Object Access Protocol (SOAP) was often used for API integration. Developers created eXtensible Markup Language (XML) messages and defined Remote Procedure Call (RPC) to execute remote operations, without the need to be aware of the implementation details.

RPCs were first proposed by Andrew Birrel and Bruce Jay Nelson in 1984 (Birrell and Nelson 1984). One of its key features was to make it easier to build distributed applications. It was extensively used as a communication mechanism in distributed systems, such as the Network Computing Architecture from Apollo Computer (acquired later by HP) (Dineen et al. 1988) and Amoeba distributed operating system (Tanenbaum et al. 1990).

In an RPC, a client program invokes a procedure on a remote server as if it were a local procedure call. When a remote procedure is invoked (Birrell and Nelson 1984), the calling environment passes the parameters across the network to the callee and the desired procedure is executed. After the execution is complete and the results are produced, these results are returned back to the calling environment, where execution resumes as if returning from a simple local procedure call.

Over time, several RPC implementations were developed and compared (Tay and Ananda 1990). More recently, Google developed a general-purpose RPC infrastructure called Stubby (gRPC 2023a). It was created to connect a large number of microservices running on their data centers for over a decade. However, in March 2015, Google opted to construct the next generation of Stubby and make it open-source: Google Remote Procedure Call (gRPC) was born. Now, the main usage for gRPC is to efficiently connect services in microservices architectures, clients (mobile devices, browsers, etc) to backend servers and create client

libraries (gRPC 2023a). Popular companies such as Netflix, Cisco and Juniper Networks are using gRPC in their systems.

Similar to other RPC systems, gRPC defines a service with methods, parameters and return types for remote calls. However, it was designed (gRPC 2023b) in a way that avoids the pitfalls of distributed objects (Fowler 2014) and the fallacies of distributed computing (Rotem-Gal-Oz 2008) while enabling interaction between services written in different programming languages. In fact, two of its core principles (gRPC 2023b) are coverage and simplicity, which state that it should be available on every popular development platform and easy to build on the chosen platform; and payload indifference, which states that it should support multiple kinds of payload types (Javascript Object Notation (JSON), XML, Thrift, etc). One of the reasons for such a feature is the definition of a language-agnostic interface using Protocol Buffers (Protobuf). Protobuf is a binary serialization format that allows an efficient and consistent data exchange between applications.

1.2 Problem Statement

The communication between services in distributed architectures in microservices has traditionally been supported by REST, due to its simplicity, generalized adoption, and easy integration with multiple platforms. However, as the systems grow in scale and complexity, relevant limitations associated with REST emerge. Those limitations become even clearer on systems that require high performance, low latency and strong consistency in service communication. Google's gRPC presents as an alternative capable of filling those gaps. In particular, gRPC outperforms the traditional REST paradigm specifically in inter-service communication in microservices architectures in key metrics such as throughput, response time, bandwidth efficiency and in bi-directional streaming (Santos 2020) (Weerasinghe and Perera 2022).

However, the adoption of gRPC in already developed REST systems is not a straightforward process. It involves technical and organizational challenges, which range from ensuring the compatibility of clients and servers during the transition to the lack of good practices. In addition, the information about how to effectively migrate REST applications to gRPC is extremely limited (Lee and Liu 2022), as there is a significant gap in academic literature regarding a methodical and strategic way to perform a migration.

Thus, it becomes important to make a comprehensive analysis of migration approaches, providing a set of guidelines that serve as a roadmap for such migrations.

1.3 Goals

The work's objective, at a high level, is to explore and compare migration approaches of REST projects to gRPC, and implement a designed strategy, providing an extensive analysis of the results based on the defined migration objectives. That is achieved by delving particularly into gRPC, but also studying other frameworks, dissecting their differences and identifying their benefits and downsides and what led to their adoption.

For the research, it is conducted in two main steps. The first step adopts the systematic literature review methodology and focuses on the current work that specifically addresses migrations from REST to RPC technologies. The second step aims to gather information

about how the industry is performing software migrations, as well as the risks and challenges they are facing. The questions that guide not only the research, but the whole work, are:

- What motivates software migrations?
- How are migrations from old systems being handled?
- Which RPC frameworks are there and what are their advantages and drawbacks?
- How does gRPC differ from REST?
- What are the main methodologies and approaches for migrating REST applications to gRPC?
- What are the most common challenges found when migrating from REST to gRPC, and how those issues can be addressed?
- Are there any tools to convert REST APIs to gRPC service? Are they effective?
- What strategies a migration from REST to gRPC can adopt from other migration types?

By doing so, the final goal is to provide helpful insights and guidelines for engineering teams studying modernization in their communication protocols, contributing to the broader discussion about the transformation and progression of distributed systems and API architectures.

1.4 Ethical Considerations

When a research in the software engineering field is conducted, the accordance to ethical considerations and standards is essential. That way, the technology created prioritizes well-being, reduces harm and promotes transparency. The ethics principles followed in research to produce the present and future materials are the IEEE Code of Ethics (IEEE 2020b) and IEEE Author Ethics Guidelines (IEEE 2020a). Below is an explanation of how specific statements of the IEEE Code of Ethics were applied.

Under Statements I.1 and I.2, the proposed guidelines and approaches encourage efficient resource usage. That way, it minimizes environmental impact and prioritizes long-term maintainability and system efficiency.

Following statement I.3, the selection of frameworks and technologies for study and/or comparison is based solely on their relevance and merit, without any external influence. There is no conflict of interest in the selection process, as all frameworks and technologies are chosen to align with the research objectives without bias.

In accordance with statement I.5, everything found in this research is based on accurate data, clearly stated assumptions and properly reviewed sources. Additionally, due credit is given to all contributions.

The migration of software applications also has ethical concerns, as it involves production systems on which users and organizations depend. In the first place, service availability and reliability need to be guaranteed. The migration should not cause any unnecessary or unjustified interruptions or degradation. Moreover, the migration must be as transparent as possible: every step and decision is properly documented to ensure the process's validity for its implementation or future work, without omitting errors, limitations and challenges.

Finally, regarding the use of Artificial Intelligence (AI) tools, their main purpose in the work was to help with text writing and grammar analysis. Other than that, every time they were used, the reason and the respective prompts and outputs were explicitly mentioned. Regardless of the assigned task, the responses were always carefully reviewed before being included.

Chapter 2

Background

The previous chapter introduced the indispensable information needed to understand the current work. This chapter dives deeper into some concepts mentioned in chapter 1 and explains their relevance.

2.1 Software Migrations

As development techniques, paradigms, and platforms advance at a faster pace than domain applications, software engineers consistently face the ongoing challenge of handling and migrating existing systems. Some reasons behind this necessity include technological obsolescence, performance and scalability improvement, cost efficiency, security matters or even the incapacity of the system to follow the organization's growth.

Software migration can be defined as "splitting and transferring software to a new platform or technology (transformation) - with the goal to meet new requirements and to improve future maintainability" (C. Wagner 2014). Obviously, the existing functionality should be retained to safeguard against the loss of business knowledge and should cause as little disruption to the existing operational and business environment as possible. However, for being such a complex process, it can quite encompass a large number of different areas of software engineering (Bisbal et al. 1998).

Depending on the application's functions, time, risks, budget and business needs, migrating software systems requires the consideration of different perspectives and factors. In certain scenarios, opting to migrate the entire application may be more suitable, while in other cases, it might be necessary to migrate specific parts of it. For that reason, there are essentially four approaches for the handling of existing systems (Althani and Khaddaj 2017):

- **Complete migration.** In the complete migration approach, a second, completely new, system is designed and developed. That includes all the functionality, interfaces and data. The migration process concludes once the target system is fully prepared and implemented, which also involves testing and transferring data from the previous system database to the target system database. Then the old system will be turned off and the new system will be activated, replacing the older one. All done in one single step. Examples of this strategy are Big Bang or Cold Turkey (Brodie and Stonebraker 1995). Although at first it may seem a good approach, as it brings a fresh design and architecture, improves quality attributes such as maintainability, performance, usability and potentially reduced costs (Althani and Khaddaj 2017), it is also associated with serious drawbacks. Some of those drawbacks can be the total time needed for the development, the necessity of skilled personnel and the high risk of failure.

- **Wrapping.** To many organizations the complete re-development of the system as whole is not an option and are forced to seek for an alternative. That alternative can be the wrapping method, as it transforms parts of the original system without changing its characteristics. It simply encapsulates the existing code into a new interface, making it easier to have access to the original software components. The advantage of this method is that software components, which have been in use and under testing for probably months or years, can be reused (C. Wagner 2014). It is much less risky and is very fast to apply (Althani and Khaddaj 2017). However, it has a short-term solution character, since the existing problems are not necessarily solved and might escalate to bigger issues. Additional steps, such as the replacement of the wrapped components, are strictly necessary.
- **Partial Migration.** The third form of software migration is partial migration, which entails transferring specific applications or components to a designated environment. It is best suited for specific business needs and is designed to ease the implementation and deployment of the migrated system. A common example of it is data migrations, where the data from older systems is processed and made available on new systems. It is both very low-risk and time-consuming, and provides a suitable solution for specific needs. On the other hand, it could not be appropriate for all applications and may not meet reliability requirements.
- **Incremental Migration.** Incremental migration is the most complex of all four approaches. Both this method and partial migration seek to find a balance between a complete migration and wrapping. Here, the new system is redeveloped into a new paradigm gradually, using incremental modifications and various development approaches. Reengineering, continuous improvement and focusing on the interoperability concept between domains (Althani and Khaddaj 2017) are some techniques used. Although it can be a complex process that requires a lot of analysis (making it highly risky), if successful, it represents a long-term solution resulting in many benefits. The process is designed to fulfill all the requirements and business needs and reduce maintenance costs by a considerable amount.

Figure 2.1 compares the referred methods, showing the ideal scenario to apply each one.

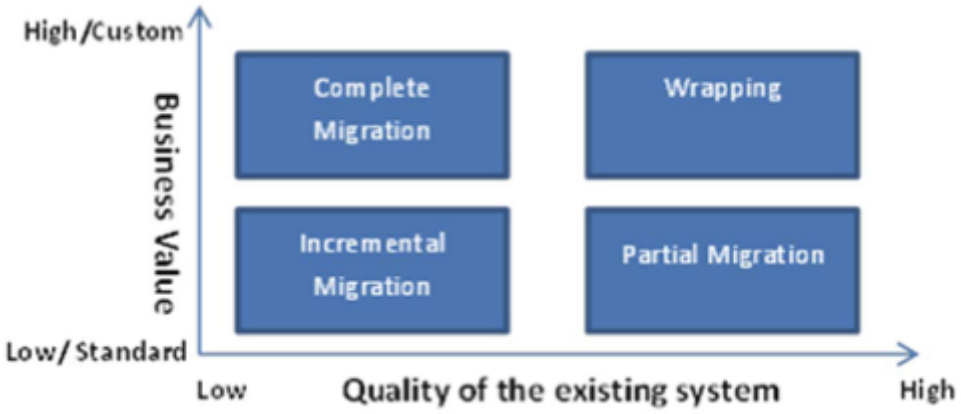

Figure 2.1: Migration approaches comparison

From Althani and Khaddaj 2017

In summary, the complete migration should be used if the existing system meets the current business needs and provides value but lacks quality. Due to the low quality, the maintenance costs of the current system are high (Almonaies, Cordy, and Dean 2010). An incremental migration should be used if the solution fails to meet the business requirements and has poor quality. Once again, the poor quality implies high maintenance costs (Almonaies, Cordy, and Dean 2010). Partial migrations are good when the system does not bring the desired value but already has the desired quality. Lastly, wrapping should be used when the value delivered meets the current business objectives and has significant quality, but the organization must implement new changes.

Although these are four distinct methods, in the end a strict separation is not possible (C. Wagner 2014). A specific activity can fit into several processes. For example, a migration can include the development of brand new components. For that reason, each part of the system should be considered individually. Anyway, regardless of the process, any migration should be separated into 5 major activities in order to be successful (Wu et al. 1997):

1. Justification
2. Previous system understanding
3. Target system development
4. Migration
5. Testing

With this, it is fair to conclude that the main drivers on conducting a software migration are the business's requirements, which are mainly determined by the risk and cost. In addition, a deep decision-making process is crucial. Weighing the advantages and disadvantages of migrations and being fully aware of the drivers and consequences is essential before starting the process. Rushing without complete consideration and investigation can lead to unplanned challenges and unmet expectations. The main goal is to achieve better quality attributes in the software, mainly reusability, expandability, performance and efficiency.

2.2 RPC Frameworks

Modern applications barely run in isolation. On the contrary, they are connected with many others through the web and communicate in order to synchronize their actions. That makes today's software systems a set of different applications running on different networks and locations. This characteristic has led to the prominence of microservices architecture, where applications are part of a collection of independent, autonomous (developed, deployed and scaled independently), business capability-oriented, and loosely coupled services (Indrasiri and Siriwardena 2018). This means that a microservices-based system requires the services to communicate through the network.

These communication techniques are mainly accomplished in two different approaches: synchronous or asynchronous. The synchronous communication style follows a request-response model. The client sends a request to a server and waits for a response. The asynchronous communication style follows an event-driven model, where services communicate asynchronously by using an intermediary - an event broker. When building synchronous-style communications, RPC can be a viable possibility.

In order to understand the key strengths and weaknesses of the RPC framework that is studied in this work, gRPC, the current section deep dives into it and compares it with a possible alternative: Apache Thrift. Apache Thrift is used for comparison due to being fairly popular and being utilized by reputable companies such as Cloudera, Facebook, Pinterest, and UBER (Apache 2023). That comparison can also be helpful to identify similarities and differences in migrations from REST to an RPC framework.

2.2.1 gRPC

As stated, RPC is a communication paradigm that allows client applications to invoke procedures or functions on a remote server. It facilitates the exchange of data and execution of code between different processes or systems. RPC enables developers to build distributed systems, where various components can interact seamlessly, regardless of their physical location or implementation details.

It follows the request-response model. The server waits for a call from a client. Once the message arrives, the server initiates its computation, extracts the procedure's parameters, executes the required operation and finally sends a response back to the caller. When the task is completed, the server returns to an idle state, waiting for future requests.

gRPC is a fairly new framework developed by Google (gRPC 2023a). As in many RPC frameworks, a contract that contains the methods, which can be called remotely, along with their input parameters and return types, is created. This interface is then implemented on the server side, which hosts a gRPC server and processes requests from clients. On the client side, it interacts with the server via stub (referred to as just a client in some languages), which contains the same methods as the server. Figure 2.2 illustrates how gRPC works. The represented gRPC clients and servers are able to run across different environments and can be written in a language that supports gRPC development.

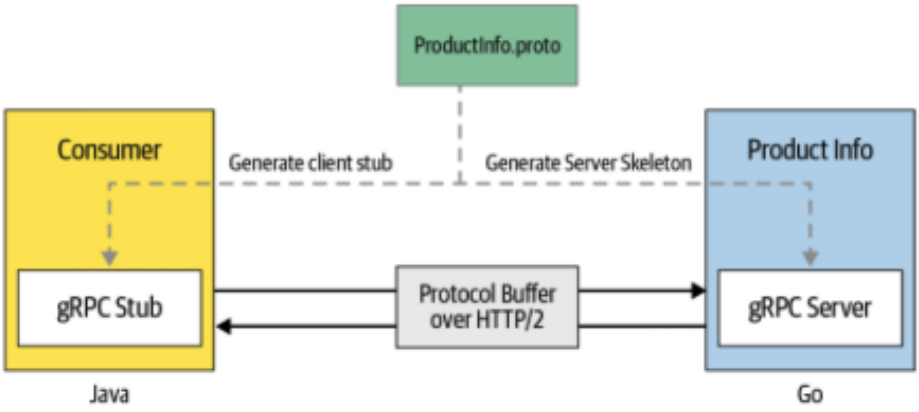

Figure 2.2: gRPC model

From gRPC 2023c

When developing a gRPC application, the first step is to define a service interface (Indrasiri and Kuruppu 2020). The service interface definitions contain the contract for the consumers: what methods are allowed to be called, the parameters for the methods and return types. The language where the service definition is specified is called Interface Definition Language (IDL). Protocol Buffers are the IDL to define the service interfaces in gRPC (it also supports

other formats such as JSON, but Protocol Buffers are the default). They are a language-agnostic, platform-independent mechanism to serialize structured data (gRPC 2023c). The service interface definition is specified in a proto file, which is an ordinary text file with a .proto extension. Table 2.1 provides a comparison between Protocol Buffers and JSON, which is commonly used in today's web APIs, to illustrate their differences more clearly.

Communication feature	Protocol buffers	JSON
Format for sending and receiving responses	Binary	Text
Platform independence	Yes	Yes
Performance	Fast	Slower, when compared to Protobuf
Flexibility	Supports certain aspects of schema evolution	Supports dynamic schema evolution

Table 2.1: Protocol buffers and JSON comparison

Using the mentioned service definition, it is possible to create the server and client code through the Protocol Buffer compiler protoc. Both simplify the logic by providing low-level communication abstractions for different programming languages. Evidently, the methods specified in the service definitions can be called remotely by the client-side. The gRPC framework manages all the intricacies typically associated with enforcing service contracts, data serialization, network communication, authentication, access control, observability, and other related aspects (Indrasiri and Kuruppu 2020). Going back to figure 2.2, the service interface definition is represented by the green block, and it will generate the white blocks, gRPC Stub for the client and gRPC Server for the server.

In gRPC the communication between clients and servers is done over Hyper Text Transfer Protocol (HTTP)/2. Much of the web today runs on HTTP/1.1. The spec for HTTP/1.1 was published in June of 1999 (Nielsen et al. 1999), almost 25 years ago. Many changes have occurred since then, making it particularly noteworthy that HTTP/1.1 has endured and thrived for such an extended period. HTTP/2, officially specified in May 2015 (Belshe, Peon, and Thomson 2015), aims to tackle scalability concerns inherited from its predecessor while maintaining a similar user experience to HTTP/1.1. It enhances the design of HTTP/1.1 in various aspects, with a notable improvement being the provision of a semantic mapping over connections.

When a gRPC client starts a service call, the gRPC client library uses the protocol buffer and marshals the data into protocol buffer format. This formatted call is then transmitted across HTTP/2. On the server side, the request undergoes unmarshaling and the corresponding procedure is invoked using protocol buffers. The response comes back with a similar execution path, flowing from the server to the client (Indrasiri and Kuruppu 2020). Marshaling is the procedure of packaging parameters and a remote function into a message packet, which is then transmitted to its destination. In contrast, unmarshaling is unpacking the message packet to retrieve the corresponding method to be invoked. (Indrasiri and Kuruppu 2020).

The popularity of gRPC has grown due to its benefits. The advantages it brings are the key to the increase in its adoption. These are the main advantages (Indrasiri and Kuruppu 2020):

- **Efficiency in process communication.** As seen, instead of making use of a text format, like JSON and XML, gRPC, by default, uses a protocol-buffer-based binary protocol for communication between servers and clients. Additionally, gRPC implements Protocol Buffers on top of HTTP/2, making it faster.

- **Well-defined service interfaces and schema.** In application development, gRPC advocates for a contract-first approach. This means it involves initially defining the service interfaces and subsequently addressing the implementation specifics.
- **Strongly typed.** As gRPC uses Protocol Buffers to define services, gRPC service contracts clearly define the types that will be used for interaction between the applications. This helps in overcoming common runtime and interoperability errors that are typically faced when applications are built by multiple teams and different technologies.
- **Polyglot.** gRPC is targeted to be compatible with various programming languages. A gRPC service definition using Protocol Buffers is language-agnostic. So, it is possible to pick any language supported by gRPC.
- **Built-in features.** In addition, it already includes support for common features such as authentication, encryption, resiliency, load balancing, among others.
- **Bidirectional streaming support.** gRPC enables both clients and servers to send a stream of messages (gRPC 2023b), allowing for more dynamic and interactive communication patterns. This is specifically valuable for scenarios involving real-time updates, as well as scenarios where efficiently handling large datasets is crucial.

However, as with any technology, gRPC has its own set of drawbacks as well. It is useful to know the downsides of gRPC in order to make an informed decision when choosing whether to use it or not. The disadvantages are (Indrasiri and Kuruppu 2020):

- **Suitability for external-facing services.** When exposing applications to external clients over the internet, gRPC could not be the most suitable protocol. The strongly typed nature of gRPC services might make the flexibility of the exposed services difficult for external consumers.
- **Drastic service definition changes.** Schema modifications are very common due to requirement changes. When there are substantial service definition changes, the code for servers and clients needs to be regenerated. This needs to be contemplated by the existing Continuous Integration (CI) process, which can include schema validations, and may complicate the overall development life cycle.
- **Small ecosystem.** The support for gRPC in browser and mobile applications is still very primitive. It is also less standardized and mature than REST, for example, which results in less variety of tools.

2.2.2 Apache Thrift

Similar to gRPC, Apache Thrift, initially created by Facebook (Mark Slee and Kwiatkowski 2007), is an open-source, multiple language serialization and RPC framework, serving as a communication platform for interactions between applications. While both frameworks have their own strengths and weaknesses, they share some common features such as support for multiple programming languages, code generation, and performance optimization.

For supporting multiple programming languages, Apache Thrift defines abstract data types in an IDL (Abernethy 2019). This IDL can be used to generate source code for any supported language. The generated code provides a complete serialization and deserialization logic for all defined types. Apache Thrift IDL also allows the definition of services in addition to those data types (Abernethy 2019). Like the data types, source code to connect clients and servers can then be generated.

Although Apache Thrift and gRPC have their own unique strengths and weaknesses, they also exhibit common features, including multi-language support, code generation capabilities, and performance optimization. This shared ground facilitates a comprehensive understanding of their main features, yet distinctions emerge when diving into their specifics. Here are the main features and how they differ:

- **Serialization formats.** Apache Thrift supports multiple serialization and deserialization data formats (Rakowski 2015), like JSON, binary, plain text, etc. That allows developers to select the format that best aligns with their specific use case. On the other hand, gRPC uses Protocol Buffers, a language-agnostic binary serialization format, which provides a smaller message size and faster processing. Both protocols are designed to be efficient and fast, but they differ in their implementation details.
- **Transport protocols.** Apache Thrift supports a wide range of transport protocols such as TCP, HTTP, sockets or files (Rakowski 2015). In contrast, gRPC primarily uses HTTP/2 as the default transport protocol.
- **Language support.** Although both support the main programming languages, Apache Thrift offers a broader range.
- **Performance.** While there are no official results comparing gRPC and Apache Thrift regarding performance, some studies have been made and they show better results for Apache Thrift (Kumar et al. 2021). However, gRPC is also being deeply benchmarked for its performance in most of its releases.
- **Documentation.** As both frameworks are fairly recent, their documentation is important. Here, Google's project has a clear advantage due to being backed by such a large corporation (Abernethy 2019)

In summary, while both Apache Thrift and gRPC offer similar functionalities for implementing RPC, there are notable differences in terms of serialization formats, transport protocols, language support, performance and project documentation. The choice between the two depends on specific project requirements and priorities. However, when it comes to bidirectional streaming and communication, Apache Thrift may not be the most natural choice. In such scenarios, gRPC stands out as a preferred choice.

Studying both gRPC and Apache Thrift provided a valuable understanding of their similarities and differences and offered a broader view of RPC frameworks.

2.3 Comparing REST and gRPC

When it comes to building synchronous request-response model communication, REST is still the most popular approach (POSTMAN 2023). Its popularity lies in its stateless architecture and support for multiple data formats. In RESTful services, the applications or services are modeled as a collection of resources that can be accessed or have their state changed by a request over the network using the HTTP protocol. The design principles of REST APIs are largely influenced by the architectural decisions of the Web, which prioritize scalability and robustness in networked, resource-oriented systems utilizing HTTP. The core principles are (Fielding 2000) (Pautasso and Wilde 2010):

- **Statelessness.** REST follows a stateless model, meaning that every request from the client should contain all the information required to be processed by the server API.

- **Resource Addressability.** Resources, which represent domain concepts, are exposed through a suitable, uniquely identified Uniform Resource Identifier (URI)
- **Uniform interface.** Resources are accessed using the verbs defined by the HTTP protocol (GET, POST, PUT, PATCH, DELETE, etc), making the API available for any client. Each method has its own expected, standard behavior and standard status codes.
- **Resource representations.** Clients are not directly aware of the internal format and state of resources; they work with representations, typically in formats such as JSON or XML, that show the intended state of a resource.
- **Hypermedia as the engine of state.** Resources as domain concepts can be related to other resources. The server response includes the URI for additional methods the client can access to discover and navigate relationships and to maintain interaction state.
- **Cacheable.** To optimize performance, an API should identify frequently called resources, cache them and set for how long they can remain in the cache.

As the number of microservices has proliferated along with their interactions, REST has fallen short of meeting the expected modern requirements. Several key limitations of RESTful services impede their effectiveness as the messaging protocol for current microservices-based applications (Indrasiri and Kuruppu 2020).

In the first place, REST is hard to impose. Although it has a lot of good practices that need to be followed in order to have a real RESTful service, they are not really enforced as part of the implementation. A report (Liskin, Singer, and Schneider 2012) conducted a study on 22 randomly picked services examining four criteria (that were useful for its evaluation). The report noted that many RESTful service implementations neglect certain aspects of REST principles. Therefore, in practical terms, a significant portion of services labeled as RESTful do not adhere strictly to the foundational principles of the REST architectural style. Consequently, many services referred to as RESTful are essentially HTTP services exposed over the network.

RESTful services are constructed on top of text transport protocols like HTTP 1.x and make use of textual formats readable by humans like JSON. The usage of such formats achieved wide adoption because of their readability and ease of debugging, the support from a great variety of tools, libraries, and frameworks, and their standardization. However, they can be particularly inefficient for service-to-service interactions.

While those formats are valuable for some scenarios, this inefficiency arises because both communicating parties don't necessarily require human-readable textual formats. On the other hand, gRPC uses a binary protocol, based on Protocol Buffers, which makes it very useful for service interactions where efficiency in terms of speed and size is preferred over human-readability.

The rising prevalence of services built with diverse polyglot technologies, interacting with each other, highlights another significant setback: the absence of well-defined and strongly typed service definitions. Most of the service definitions for RESTful services are not tightly integrated with the underlying architecture or messaging protocols. That is the case of, for example, OpenAPI/Swagger, which serves more as a supplementary component, often used for documentation or testing purposes rather than being integral to the core functionality

of the service or its communication protocols. For this reason, having a modern strongly typed service definition technology and a framework that generates the core of the server and client-side code, like in gRPC, has proven to be very useful. It can prevent many incompatibilities, runtime errors, and interoperability issues in building such decentralized applications.

Chapter 3

State of the Art

This chapter analyses the elements and concepts that are considered most significant to allow a contextualization of the state of the art. It begins with a section of scientific and peer-review literature, showing which literature review methodology was chosen and how it was conducted, followed by a section with practical developments and insights published by the industry.

3.1 Literature Review

This section explains the work's literature review. It begins by presenting the literature review methodology and its planning, followed by the conduct of the research and the conclusions and results gathered from the study.

3.1.1 Methodology

Conducting a literature review is an essential step of a research, promoting a good understanding of the state of the art and identifying gaps and obstacles on the subject. A systematic literature review is a research methodology that establishes a structured set of steps to methodically organize the review. Not only that, but it also provides a reproducible method for identifying, selecting, and evaluating literature relevant to the field while offering a reliable foundation for addressing the targeted research questions (shown in the next sections) and suggesting areas for further investigation.

Additionally, as highlighted in the problem statement section, the availability of scientific and reliable information on how to migrate REST applications to gRPC is very limited. For that reason, in addition to the systematic literature review, carefully selected grey literature are also included to ensure a broader and more comprehensive analysis. This includes technical blogs from reputable companies and official documentation from the used frameworks.

Following the guidelines from (Kitchenham and Charters 2007), a method was elaborated to perform a systematic review, divided into three main steps: planning, execution and conclusions.

Having a clear plan to ensure a literature review's reliability and reproducibility is essential before proceeding with a systematic literature review. That is why the first step of the review is the planning phase. It is of crucial importance because it defines the scope and criteria that guide the whole review process, with one of its most important activities being the definition of the research questions (Kitchenham and Charters 2007). The subsequent two subsections detail the components of the planning phase, which include the Population,

Population	Enterprises and development teams interested in improving service performance and efficiency
Intervention	Migration from REST-based applications to gRPC; adopt protocol buffers instead of human readable playload formats for data serializations; study tools and frameworks that help in the migration
Comparison	Compare other common migration types and approaches in the software industry; understanding how gRPC can benefit software systems
Outcome	Tools and best practices to perform a migration; identify common challenges during the migration
Context	Distributed applications communicating synchronously through REST

Table 3.1: PICOC Framework Defining the Population, Intervention, Comparison, Outcome, and Context for the Systematic Literature Review on Migrating from REST to gRPC

Intervention, Comparison, Outcome, Context (PICOC) framework, research questions and review protocol.

3.1.2 Research Framework and Questions

The research questions are an important part of the planning of a systematic literature review. They must be well thought through and phrased as they set the focus for the identification of the primary studies, the extraction of data from the studies, and the analysis (Wohlin et al. 2012). The questions are framed recurring to PICOC criteria (Kitchenham and Charters 2007), which stands for Population, Intervention, Comparison, Outcome, and Context. By using this framework, the questions are aligned with the objectives of the investigation, ensuring a clear and methodical approach. The application of the PICOC framework to this study can be found in Table 3.1.

In order to ensure that the study is relevant and applicable to real-world and practical scenarios, the population chosen was enterprises and development teams interested in improving the performance and efficiency of their services. This should be the teams responsible for planning and executing the migration. This population allows the research to address the needs of developers who currently rely on systems built using REST.

The intervention is the focus of the research since it is the procedure that tackles a specific issue (Kitchenham and Charters 2007). At a high level, the intervention for the research is the migration from REST-based applications to gRPC but it also includes other sub-parts of it, such as adopting protocol buffers instead of human-readable payload formats for data serialization and studying tools and frameworks that facilitate the migration process.

Regarding the comparison, it is made against other common migration types and approaches in the software industry as well as understanding how gRPC benefits software systems compared to REST or other communication protocols. The comparison between gRPC and other approaches points out its advantages and limitations and provides a balanced perspective.

For the outcome, it should relate to factors of importance to the population (Kitchenham and Charters 2007). That is why the focus of the outcome is to provide practical tools and best practices to perform the migration from REST to gRPC. This way, it helps minimize

Keywords	Related keywords
Migration	Modernize; refactor; transition
REST	REST API; REST-based; application; RESTful services; RESTful systems
gRPC	Remote Procedure Call; RPC; Apache Thrift

Table 3.2: Keywords extracted from PICOC and their synonyms

downtime - a key concern in maintaining service availability. Additionally, identifying common challenges during the migration helps teams anticipate risks, ensuring a smoother adoption of gRPC without unnecessary disruptions.

Finally, the context is simply distributed applications communicating synchronously through REST as most modern software relies on information and integration from other software systems.

From the aspects outlined in the PICOC framework, various key questions come out regarding the migration from REST to gRPC. These questions focus on the methodologies used, the challenges faced, strategies to minimize disruption, the effectiveness of migration tools, and insights from other migration types. Thus, the research questions were formulated:

- What are the main methodologies and approaches for migrating REST applications to gRPC?
- What are the most common challenges found when migrating from REST to gRPC, and how can those issues be addressed?
- Are there any tools to convert REST APIs to gRPC service? Are they effective?
- What strategies a migration from REST to gRPC can adopt from other migration types?

3.1.3 Review Protocol

With the research questions defined, a review protocol is needed to minimize bias while capturing studies that answer those questions. The purpose of a review protocol is to specify the methods that are used when performing a systematic literature review (Kitchenham and Charters 2007). For the current research, the protocol includes the development of the search string derived from the PICOC framework, the selection of academic databases, the inclusion/exclusion criteria to filter studies and the selection process.

Applying the PICOC framework to the research allowed the extraction of vocabulary to ensure a comprehensive coverage of relevant literature. Furthermore, to capture variant phrasings and expressions in the literature, each keyword was expanded by a few equivalent synonyms. The words extracted and their corresponding synonyms can be found in Table 3.2.

Other keywords/synonyms were extracted but they were omitted as they didn't improve the search for studies. That was the case of protocol buffers, data serialization and distributed systems, among others.

With the keywords and synonyms defined, developing a search string becomes an easy task. The search string created was:

Inclusion criteria	Exclusion criteria
Books, conference papers, journals, technical reports/magazines, master's/doctor theses	Not relevant to at least one research question
Sources written in English or Portuguese	Prior to 2015
	No real case studies

Table 3.3: Inclusion and exclusion criteria

(rest OR restful OR rest based) AND (migrat* OR moderni* OR refactor*) AND (gRPC OR remote procedure call OR rpc OR Apache Thrift)

This search string was then refined in order to be adjusted to the academic database search capabilities. The databases chosen to conduct the literature review were IEEExplore, ACM Digital Library and ScienceDirect. Those terms were also only applied to the abstract as a means to avoid unrelated sources and since abstracts are the richest source of relevant keywords (Shah et al. 2003).

The literature queried by the search string then goes through a triage process based on the inclusion and exclusion criteria defined. Those criteria, once again, guarantee the relevance and quality of the selected studies and make sure they are aligned with the research questions. Table 3.3 presents the inclusion and exclusion criteria applied to the review.

The inclusion of books, conference papers, journals, technical reports/magazines, as well as master's and doctoral theses, aims to collect the maximum available data by considering not only sources published in the science community but also other sources that might contain relevant findings. Opting for texts written in English or Portuguese is due to the necessity of correct analysis and interpretation of the sources. Regarding the exclusion criteria, it was chosen not to include studies that do not show any relevance to the research to keep the focus on the subject. It also excluded sources prior to 2015, as gRPC was only made available in that year. Finally, studies that did not present real case studies were excluded due to the practical applicability nature of the current research.

For the selection process, it starts by checking the source's titles. If it is not enough to determine whether the study is relevant or not, then the abstract is reviewed. If the abstract seems relevant to the research, the full source is examined based on the inclusion criteria.

Now that the review protocol is done, the review execution can start. From here, the process followed to conduct the review is documented, according to the protocol defined. It includes the identification of research, study selection, extraction and synthesis of data and quality assessment of selected sources.

3.1.4 Identification of Research

The main goal of this section is to identify relevant primary studies in a systematic way, while avoiding a biased search (already highlighted). For that reason, to ensure the research transparency and replicability (as far as possible), the search is fully documented. The documentation process has the name of the database, the search strategy for that specific database, the date of search and the years covered by the search (Kitchenham and Charters 2007). Additionally, it includes how many studies were returned by the database after applying the strategy.

The search was executed in the academic databases IEEExplore, ACM Digital Library and ScienceDirect, between the 22nd and 23rd of March 2025. The search string, built from the terms in the PICOC framework and respective synonyms, was adapted to match the search requirements of each database. The results were exported and there was no need for duplicate deletion as there were no duplicates. Table A.1 in Appendix A shows in detail the terms used and results from each database.

Specifically to the ScienceDirect database, as its search engine only supports a maximum of eight boolean connectors, the search string was divided into three different queries in order to get as much data as possible.

3.1.5 Study Selection

With the primary studies gathered, they now need to be assessed for their actual relevance to the research. This is where the inclusion and exclusion criteria are applied.

In an initial stage, the easiest criteria to check were applied: those requiring only metadata examination instead of an analysis of the full text. It included the publication type, publication date and text language, according to all the inclusion and one exclusion criteria (prior to 2015). No studies were excluded here.

Moving on to analyze the title and abstract, a single study was excluded as it fit in the "not relevant to at least one research question" exclusion criteria. All the others, not only passed all the exclusion criteria but also showed to be highly relevant to the research. That thought becomes even more clear after reading the full studies.

That means that a total of two studies were included in the final revision. In Appendix A, Table A.2 summarizes the selection process done over the identified studies. Although it is recommended to just keep the record of studies that went through a more detailed inclusion/exclusion criteria (Kitchenham and Charters 2007), due to the low number of candidates primary studies, all were included.

3.1.6 Quality Assessment and Data Extraction

Besides the inclusion and exclusion criteria, it is also very important to assess the quality of primary studies. Especially for this research, it weighs the importance of studies when the results are being synthesized and guides the interpretation of findings - in summary to assist in data analysis and synthesis. For this reason, the quality assessment and data extraction forms are presented and fulfilled simultaneously (Kitchenham and Charters 2007).

In spite of "quality" being a complex and hardly defined concept, the quality assessment of a study should contemplate (Tacconelli 2009), among other aspects not so relevant to the current research:

- Appropriateness of study design to the research objective
- Risk of bias
- Choice of outcome measure
- Quality of reporting
- Generalisability

Each study is evaluated using the Critical Appraisal Skills Programme (CASP) (CASP 2024) checklist, specifically developed for qualitative research. It consists of a set of questions, which are answered with "Yes", "No" or "Can't tell". Once the checklist is fulfilled, if there are a large number of "Can't tell", it should be considered if the conclusions of the study are reliable and carefully interpret the results.

Regarding the data extraction form, it includes the title and year, the context, objectives, methodology, data collection and analysis, conclusions and limitations. Some fields, such as author(s) and document type, among others, often found in data extraction forms, were excluded as they were already presented.

Appendix A shows a Table (A.3) with the quality assessment checklist applied to both selected sources, as well as the data extraction form properly filled.

3.1.7 Data Synthesis

Finally, the data synthesis is the last step of a systematic literature review, where the results and conclusions of the primary studies are summarized. The synthesis for the current review is descriptive.

One of the first conclusions from the review highlights what has already been said in the problem statement: the information on how to migrate REST applications to gRPC is very limited. That is proved by the lack of scientific and peer-reviewed sources, with only a single study addressing the subject directly. The search string presented went through several changes and refinements, but in most cases, the academic databases returned either sources not relevant to the research or no sources at all.

Some questions regarding the quality assessment, related to data collection and analysis, were answered with "No" as the paper focuses on the development and demonstration of a refactoring method rather than describing data collection and analysis. For the "Can't tell" answers with respect to ethical considerations, it was chosen as many sources do not explicitly state that ethical considerations were taken into account. However, the absence of such information in both studies does not allow to safely conclude if those considerations were not followed or just omitted in the document. Finally, based on the conducted review, it is now possible to consider how the studies answer or give insights into the research questions.

What are the main methodologies and approaches for migrating REST applications to gRPC?
It is commonly agreed that one of the first and main steps of a migration approach is to identify all the entry points in an API and have a clear understanding of the data being exchanged, mainly their parameters and return types. Following a coding convention facilitates their identification, whether it is done manually or through automation. Once the endpoint in a REST application is identified, the protocol buffer files should be written and the code generated. The next stage is to refactor both the REST server and client code and finally validate the refactored system. Wherever automation is possible during this process, it should be implemented.

While this approach is simple and straightforward, it may face some challenges due to code ownership. The approach requires both the client and the server code to be refactored at the same time, as they will have incompatible protocols if not. In case a team does not own one of the parts, or at least is not able to change it, the migration can (and most likely will)

be blocked until both teams make arrangements to perform the migration. This requires more organization, communication and probably costs.

What are the most common challenges found when migrating from REST to gRPC, and how can those issues be addressed?
No challenges related to the migration were directly reported by the sources. However, that emphasizes the care that must be taken when identifying the API endpoints, as some can go unnoticed and cause unwanted errors. During tests, if such a scenario happens, the endpoint should be identified again and updated. It is recommended to use Graphical User Interface (GUI) client tools during and after the migration to avoid those cases.

Additionally, the fact that gRPC is designed to be compatible with multiple programming languages makes the process of automating some steps even more difficult, mainly in the endpoint mapping. The reason behind that is the differences in language syntax, conventions, and behavior. An example of a simple controller in two different but similar languages, Java and C# (Listing 3.1 and 3.2), is shown.

```java
@RestController
@RequestMapping("/api/appointments")
public class AppointmentController {
  @PostMapping
  public ResponseEntity<String> bookAppointment(@RequestBody
    AppointmentRequest request) {
    return ResponseEntity.ok("Booked Dr. " + request.getDoctorName() + "
    on " + request.getDate());
  }
}
```

Listing 3.1: REST controller in Java

```csharp
[ApiController]
[Route("api/[controller]")]
public class AppointmentController : ControllerBase {
  [HttpPost]
  public IActionResult bookAppointment([FromBody] AppointmentRequest
    request) {
    return Ok($"Booked Dr. {request.getDoctorName()} on {request.getDate
    ()}");
  }
}
```

Listing 3.2: REST controller in C#

As seen in both examples, there are many key aspects that differ from one language and framework to the other. For example, the route definitions are different: Spring requires explicit paths through `@RequestMapping` annotation, while .NET uses `[Route]` with token replacement `[controller]`, auto-generating routes from the class name. There are also differences in the method signature. Spring wraps the response type in `ResponseEntity`, uses camel case for the method name and `@RequestBody` for request body binding. On the other hand, .NET uses `IActionResult` for the response type, Pascal case as method naming convention and `[FromBody]` for request body binding. There are more divergences between these two frameworks, but for the objective of this analysis, there is no need for

such a deep dive. However, to summarize, the more languages or frameworks supported by an automation tool, the more difficult it is to develop such a tool.

Are there any tools to convert REST APIs to gRPC service? Are they effective? From the information available in the research, there are no tools to perform a migration from REST applications to gRPC. Nevertheless, as stated, it is suggested to use GUI client tools for testing purposes. One that is specifically mentioned is BloomRPC (https://appimage.github.io/BloomRPC), which supports gRPC message types.

A tool not mentioned by the sources, but that might prove to be useful (according to what was reported by them) is an API description tool. A REST API fully documented in an API description tool, such as Swagger, POSTMAN, Redoc, should facilitate the task of identifying the endpoints and their in and output data.

What strategies can a migration from REST to gRPC adopt from other migration types? Besides the aspects that have already been highlighted in the current analysis, for instance, the automation of migration steps and identification of endpoints, the sources revealed other strategies common in other migration types that can be applied in the migration from REST to gRPC.

That is the case of regression testing. During software upgrades or changes, it is highly recommended to validate that the refactoring process did not insert unintended changes or errors. The same can be applied in the migration under study. Additionally, a phased rollout helps in minimizing risks, ensures the changes work as expected and that the system is stable. It is a common practice when releasing new features and in risky and complex migrations, such as those from monoliths to microservices.

3.2 Industry Migrations

As shown in the preceding section, when searching for scientific and peer-reviewed work on how to migrate REST applications to gRPC, very few results relevant to the work were returned. As an alternative, despite not following the academic, company blogs and conference talks often offer useful and trustworthy information about certain topics. This section presents articles published by companies/engineers with a recognized reputation in the software engineering community. By applying such filter to the chosen articles, the goal is to ensure the credibility of the information and complementing at the same time the literature review previously conducted.

3.2.1 WePay

In 2019, WePay published an article on their blog on migrating APIs from REST to gRPC (WePay 2019). WePay is an online payment service provider based in the United States. The company gained recognition for its easy integration through APIs and partnerships with various platforms, including crowdfunding and event management sites. The blog of a company of such magnitude was considered fairly credible.

The requirements to perform the migration were (WePay 2019):

- Minimize the amount of reword
- Not changing the current pipelines for build and deploy
- Non-migrated clients should be able to communicate with migrated servers

- gRPC should comply with the same base platform as REST microservices (for exposing health metrics, for example)

The migration approach was to run the gRPC server as a thread within the existing application infrastructure. A shared library to plug into the migrated microservices was also built in order to comply with the base platform already in use for REST applications. No further details were provided on how or when to discard RESTful services, and whether all the endpoints were migrated at once or not.

An alternative approach would be to use gRPC-Gateway (gRPC-Gateway 2023). A gRPC-Gateway reads a gRPC service definition and creates a reverse-proxy server (gRPC-Gateway 2023). This server translates a RESTful JSON API into gRPC, allowing non-migrated clients to keep communicating with migrated servers.

The team at WePay decided to use a central git repository to store all Protocol Buffer files (WePay 2019). This decision promotes consistency and standardization across all services, serving as a singular source of truth for Protocol Buffers definitions. Moreover, having all files stored in a single repository can be potentially useful for migration purposes as it will be visible when a service has already been migrated to gRPC. Figure 3.1 shows the full lifecycle for gRPC services at WePay.

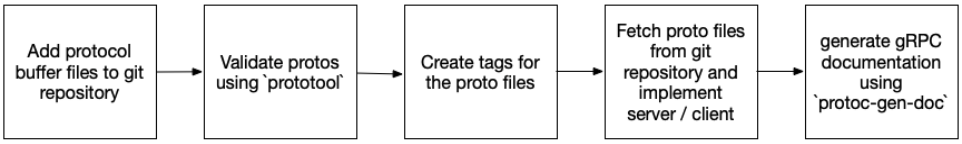

Figure 3.1: Lifecycle of gRPC at WePay

From WePay 2019

An approach for refactoring REST services to gRPC has also been proposed (Lee and Liu 2022). However, it was discarded since it requires that both servers and clients be migrated simultaneously. This constitutes a problem for systems whose services are under the ownership of different teams and might have a large (or even an unknown) number of clients.

3.2.2 Google Cloud

With a crucial role in the development and evolution of cloud technologies, Google Cloud has been a reputable reference in that area for some time. Its direct connection to gRPC, a framework also created by Google, as mentioned in section 1.1, makes their blog posts especially relevant to the work.

Recently, they published a post on how to make gRPC services available to gRPC clients (Strebel and Tahouri 2024). It is admitted that most inter-service communications are still done via REST and, even though gRPC is seen as a high-performance communication protocol, changing those APIs would require both notable effort and investment of resources as well as discourage engineers who lack the knowledge and experience working with the framework.

For that reason, to fill the significant gap between gRPC services and RESTful APIs, it is proposed the use of an open-source tool. The tool uses a gRPC gateway to generate an adapter for HTTP requests. It translates those incoming requests to gRPC, making

the co-existence and integration with REST systems much easier and, more importantly, facilitating the work of teams planning a transition from REST to gRPC.

The mentioned gateway is based on a proto file, which defines the services and message types, working as a contract between the service and its consumers. Additionally, the team also created a wrapper to generate the gateway by referencing the proto file, so that the process of generating the gateway is simplified. It also wraps the gateway in a Go module to be deployed independently.

However, it still has some flaws. The API does not follow widely accepted REST principles and does not offer common features typically found in APIs, such as authentication, monetization, monitoring, and error handling, among others. Developers also don't have access to a self-service that enables the discovery of news APIs. To achieve some of those capabilities, an API management solution was used.

3.2.3 LinkedIn

With a massive distributed architecture and notable technical knowledge, LinkedIn is often seen as a reference in the software industry. LinkedIn has developed Rest.li due to discontent with existing REST frameworks. However, Rest.li has several limitations that could be mitigated by gRPC. Therefore, a migration plan was made to fully move to gRPC (Ramgopal and Chen 2024).

In order to have a smooth transition, the migration is done in two highly automated phases by relying on a bridge to connect both frameworks, called Alcantara. The objective of Alcantara is to expose both Rest.li and gRPC interfaces, allowing a gradual and controlled migration and keeping Rest.li and gRPC running simultaneously. That way, no production services were interrupted and no manual work from developers was needed.

The first phase is divided into two steps: the migration of the data models, written in Pegasus Definition Language (PDL), and the migration of the API contracts, written in the Rest.li IDL. For the first step, a tool was developed to convert the PDL schema to protobuf. From the moment they are converted, developers should work on the protobuf schema, as there is a reverse translator to keep compatibility with existing mechanisms. Additionally, an automated validation was created to check for differences between the two models. The second step follows the same logic as the previous step, but by applying it to Rest.li IDL: a protobuf is automatically generated from the IDL and the developers start to evolve the protobuf schema since the changes to the protobuf schema were propagated to the IDL.

Now that the services are migrated, the second phase is about handling the clients, as clients are still using Rest.li. After the first phase of the migration, the services start exposing two endpoints: one for Rest.li and other for gRPC. In order to avoid rewriting the clients, an abstraction layer, called Rest.liOverGrpcClient, was introduced. It is responsible for checking if the service has already been migrated. If that's the case, it automatically converts the Rest.li call to gRPC. Otherwise, the usual Rest.li call is done. Now, it is possible to slowly start to retire clients and, once it is done for all clients, the servers can start to be retired.

It is acknowledged that some incidents were related to the migration; however, the use of tracing and alerting tools has helped with debugging.

3.2.4 Lessons learned

All this industry evidence shows that the migration from REST systems to gRPC requires a notable technical and organizational challenge, mainly in large distributed systems. However, they also demonstrate that with proper planning, a structured approach and the right tools, the transition can be done gradually, safely and, most importantly, without compromising the systems running in production.

Although there are differences among the cases shown, it is still possible to identify common approaches and valuable lessons:

- **Gradual adoption** - Both WePay and LinkedIn went for a phased migration, allowing the coexistence of REST and gRPC during the process. With such an approach, the risk is minimized and makes the transition easier by removing the need of rewrite all the services and clients at once.
- **Bridge tools** - The usage/creation os bridge tools, like Alcantara in LinkedIn and gRPC Gateway in Google Cloud, has shown to be essential to keep the compatibility between clients and services during the migration.
- **Automation and code generation** - The automatic generation of interfaces and validators has been revealed to be a key factor in reducing the manual effort and ensuring the consistency of all systems. LinkedIn approach is highlighted by totally automating the conversion of PDL/IDL to protubuf.
- **Central artifact repository** - WePay's decision to centralize all profobuf files in a single repository shows the importance of visibility during the migration process, facilitating the identification of services already migrated.
- **Limitations** - Although gRPC adoption brings a lot to the table, limitations have been found. Google Cloud, for example, recognizes the absence of certain features common in REST APIs, which can make a migration difficult when being dealt with by teams with less experience.

Summarizing, all these real-world experiences complement the literature review. Although there is no widely agreed-upon or trivial approach to migration guidelines, the analysis of all these practical cases showed common patterns and strategies validated by the organization's know-how and experience. And, even though they all have different realities and contexts and faced distinct challenges, it is clear that the adoption of gRPC in REST applications requires proper planning, mainly in distributed architectures maintained by a large number of teams.

Chapter 4

Experimental Design

After the state of the art and industry migrations review, a plan was designed in order to carry out experiments and apply the knowledge gained. This chapter describes the methodology, tools and the project adopted to perform a migration from a REST-based system to gRPC.

4.1 Methodology

Now that the knowledge has been acquired through the systematic review and industry cases analysis, the goal is to apply it in a real-world context by performing a migration of an existing REST application to gRPC. This practical exploration allows to directly face the technical challenges related to the transition between the two communication paradigms, as well as observe the efficiency of strategies and good practices identified.

The choice of using an existing project makes the reproduction of conditions often encountered by engineering teams possible, where production systems with functional and non-functional requirements are already defined and need to be modernized without compromising their stability. The goal, beyond performing the migration itself, was to also understand which aspects make the migration process easier or harder, which decisions are the most critical and how mechanisms to promote compatibility, maintainability and reduce the effort can be created (or leveraged if already created).

The adopted approach has an iterative nature, inspired by the data gathered from the literature review and the patterns observed in real cases such as WePay, Google Cloud and LinkedIn. The automation of repetitive tasks and integration with proto contracts were taken into account and the migration was conducted in a way that maximizes simplicity, maintains compatibility with existing clients and aligns with modern software best practices.

Throughout the whole process, everything was documented, from the main migration steps to technical decisions and challenges faced, in order to extract valuable insights and potential generalizable knowledge. All the experience obtained served to validate the approaches studied, but also propose improvements, identify gaps and recommend practices that can be useful in similar migration processes.

4.2 Migration Objectives

The clear definition of objectives is an essential step to evaluate the success of the migration. The chosen objectives were designed in a way to balance simplicity, service availability and preservation of existing functionalities.

In the first place, the migration must be conducted in the simplest way possible, avoiding unnecessary complexity in both technical and organizational processes. Such simplicity not only avoids the risk of introducing errors but also allows the strategy to be replicated or adapted in similar contexts. It also includes the system compatibility with non-migrated clients.

Another objective is to keep the service downtime as low as possible. For that reason, it is mandatory that the interruption associated to the migration process is not more than the time required for a regular release of the system prior to the migration. This ensures that the transitions can be done in conditions compatible with the current operations, with an impact on the end users.

Additionally, the preservation of existing functionality is also a requirement. The migration must not introduce unwanted behavior or capabilities loss, ensuring the functional equivalence between the original system and the migrated system.

Lastly, although the current work does not include any performance tests, it is assumed as a migration objective the improvement of the application's performance. One of the main reasons for choosing gRPC over REST is precisely the performance advantages that it brings, as considered in Section 1.2. Hence, after a migration, if the system's performance has not improved, it can be hardly said that the migration was successful.

4.3 Project Selection

The project to migrate plays a crucial role in the work, as its characteristics directly affect the viability and, more importantly, the relevance of the migration. For that reason, before starting both the migration process and the project selection, it is essential to define a set of criteria that guide the selection of an adequate project.

In the first place, the project should be open-source. Not only that, it should also have a valid license so that it is free to use, change and distribute. Common open-source licenses include MIT, Apache 2.0, GNU GPL and BSD licenses. Other than that, it must be implemented in a programming language familiar to the author, ensuring its technical analysis and a well-founded migration. Very small projects should be avoided, as they are too simple, as well as massive applications, as they are too complex to focus on a migration.

Ideally, it should have recent releases, active issues, regular commits and an overall solid contributor base. Good documentation about the architecture, components and how to set it up is also highly valuable. Although it is not mandatory, a well-defined structure and a good set of test cases can certainly make the migration easier. Obviously, it should follow the REST principles.

Despite not meeting all the criteria listed above, a project that is a good fit is the Sprint Petclinic, particularly the microservices version (Community 2025a) that is part of the Sprint PetClinic Community.

4.3.1 About the Project

The Spring PetClinic (Community 2025b) was created in 2003 by some of the authors of the first version of the Spring Framework to be a sample application of what it could do at the time. As time passed, the Spring ecosystem grew, but Spring PetClinic didn't. However, it revived in 2013 and in 2016 all forks of it were centralized in a single GitHub organization,

offering several variants of the initial projects, such as the microservices one. The Spring PetClinic Microservices is the distributed version of the Spring PetClinic sample application.

Functionally, the project simulates the operation of a veterinary clinic. The main screen can be seen in Figure 4.1.

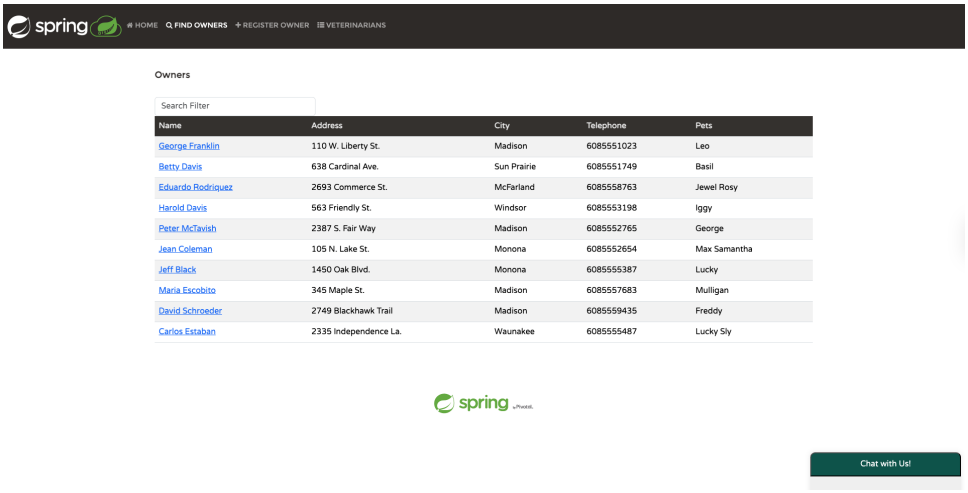

Figure 4.1: PetClinic main screen

The UI does have much complexity: on top, there's a navigation bar to guide the users through the application and the information is displayed in simple tables and added in basic forms. The main functionality evolves around adding, editing and visualizing information about the clinic's customers. Each customer may have multiple pets and it is possible to register their data. Associated with pets, it is also possible to register visits with information about date, appointment description and clinic history. To easily find any of this data, the project offers a simple text input to search for information related to the customer and pet. Other than that, it allows viewing the veterinarians available, including their specialties. Finally, it integrates a chatbot, allowing users to interact with the application in natural language.

4.3.2 Initial State

The Spring PetClinic Microservices is a microservices-based implementation of the classic Spring PetClinic, demonstrating modern distributed architecture patterns. It is written in Java and uses Spring Boot and Spring Cloud. The system consists of multiple independent services that communicate via REST APIs. The core components are:

- **Customers Service** - Manages customer data
- **Vets Service** - Manages veterinaries data
- **Visits Service** - Manages veterinaries visits data
- **GenAI Service** - Provides a chatbot
- **API Gateway** - Routes client requests to the services and handles the user interface
- **Config Server** - Manages the central configuration for all services
- **Discovery Server** - Eureka-based service registry

Figure 4.2 shows the architecture of the system as a whole.

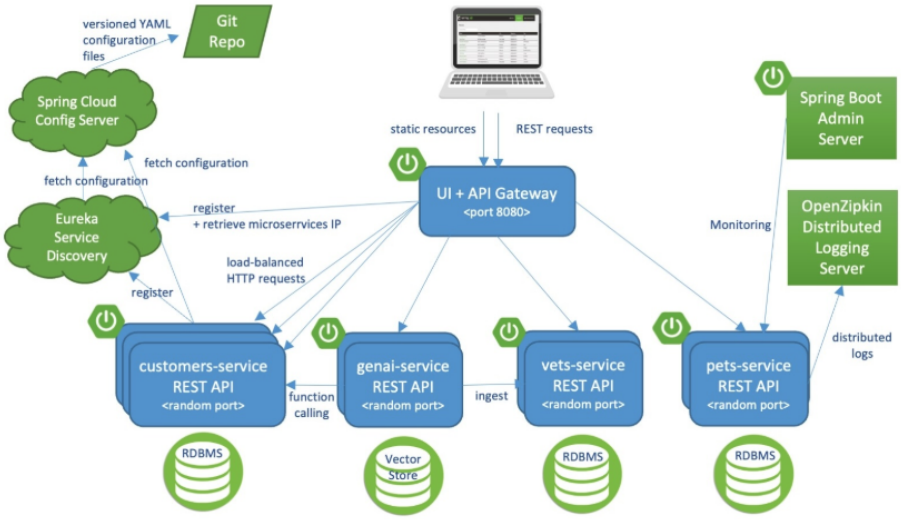

Figure 4.2: Spring Petclinic Microservices architecture

From Community 2025a

Each microservice follows a layered architecture, organizing it into distinct horizontal layers, each with specific responsibilities and having unidirectional dependencies. Layered architecture (Savolainen and Myllärniemi 2009) has been one of the first architectural styles ever used and it is still very common today, despite having very different implementations. From top to bottom, the structure of each service is composed of a presentation layer, a service layer and a data access layer.

The presentation layer is the entry point to the service. It handles HTTP requests, request/response mapping and validation. The service layer is responsible for the business logic. This layer is very lightweight as the business logic of the domain is minimal. For that reason, the controllers interact directly with the data access layer. The data access layer handles persistence by having the domain entities and the repository implementations. All that makes each service have a clear separation of concerns by package structure and the dependencies between packages are unidirectional.

On the other hand, the analysis of the project also revealed its poor testing, with some services not having tests at all. For example, the test coverage is considerably low. Table 4.1 shows the test coverage for each service.

Service	**Test Coverage**
Visits	66%
Vets	57%
GenAI	No tests
Customers	41%
API Gateway	83%

Table 4.1: Spring Petclinic Microservices test coverage

The truth is that code coverage, by itself, does not prevent a high percentage of faults (Hemmati 2015), as it does not directly measure the quality of tests. Nonetheless, assuming

that the covered parts are well tested, the effectiveness of the tests suits is still weak. That happens not because the tests are poor, but because there's too much code that remains untested.

The API Gateway shows the highest coverage among all services, with 83% of code covered by tests. Although it has the highest value, it is only slightly above what is acceptable, as many teams have a minimum threshold of 80%, indicating a bigger effort in testing the parts exposed to the outside. On the other hand, the more central components of the system, such as customers, visits and vets, present significantly lower values, having 41%, 66% and 57% respectively. The GenAI component did not contain any tests implemented by the time of the analysis.

This discrepancy in test coverage percentage levels not only highlights the difference in effort required to ensure the quality of all components but also introduces additional challenges to the migration: the absence of automated tests in some components makes it more difficult to maintain functional non-regression after changes to the communication protocol.

Continuing the analysis, the project does not have a contract definition for services. They communicate via REST calls; however, the structure and parameters of the request and response bodies were not described in any specification file, such as OpenAPI or Swagger. For this reason, the understanding of each API depended solely on the analysis of the source code.

On top of that, no explicit version was exposed on the endpoints. Any changes made to the API were applied to the existing resource addresses, without distinction between versions. It does not represent a significant issue in the Spring Petclinic Microservices project context because it is a simple sample application. However, in contexts where an API has multiple consumers or different teams, the absence of versioning increases the difficulty of service evolution and the risk of introducing incompatibilities.

The transition to gRPC presents an opportunity to introduce a more contractual approach, supported by proto files that serve as the single source of truth, facilitating the controlled evolution of APIs.

Chapter 5

Migration

Once the methodology and design have been defined and the project has been selected and studied, the migration needs to be executed. This chapter provides a detailed description of the migration process from the original system, based on a REST architecture, to a new communication model supported by gRPC. Throughout the chapter, the main decisions, the steps followed on each migration phase, and every mechanism used are presented.

5.1 Preparing the Migration

Before starting the migration, it is necessary to perform a series of preparatory tasks to ensure that the process unfolds in a structured manner and is coherent with the objectives. One of these tasks is to identify the existing services and how they interact with each other. In fact, that is one of the initial steps of a migration, as seen in the literature review. To perform such a task, the code was manually analysed. Figure 5.1 shows the interactions between services.

With all the APIs entry points found, it is relatively easy to understand the inputs and outputs. From now on, it is possible to clearly see which operations are exposed and the data needed to execute them. This not only serves to understand the system's structure, but also forms the basis for defining gRPC contracts, making it possible to extract the relevant data to form the services and messages in proto files.

Another task to do for preparation purposes is to install all the necessary tools for the migration. Besides everything needed to compile and run the project, the installed tools were:

- Postman - Testing GUI used to call both the REST API and gRPC services
- Protocol Buffer Compiler (protoc) - Compiling proto files, which contain service and message definitions
- Protoc Gen gRPC Java - Plugin of Protocol Buffer Compiler to generate Java code from proto files
- Go - Language in which the gRPC Gateway is written
- gRPC-Gateway - Plugin of Protocol Buffer Compiler to translate RESTful HTTP API calls into gRPC
- protoc-gen-go - Plugin of Protocol Buffer Compiler to generate Go code from protobuf message definitions

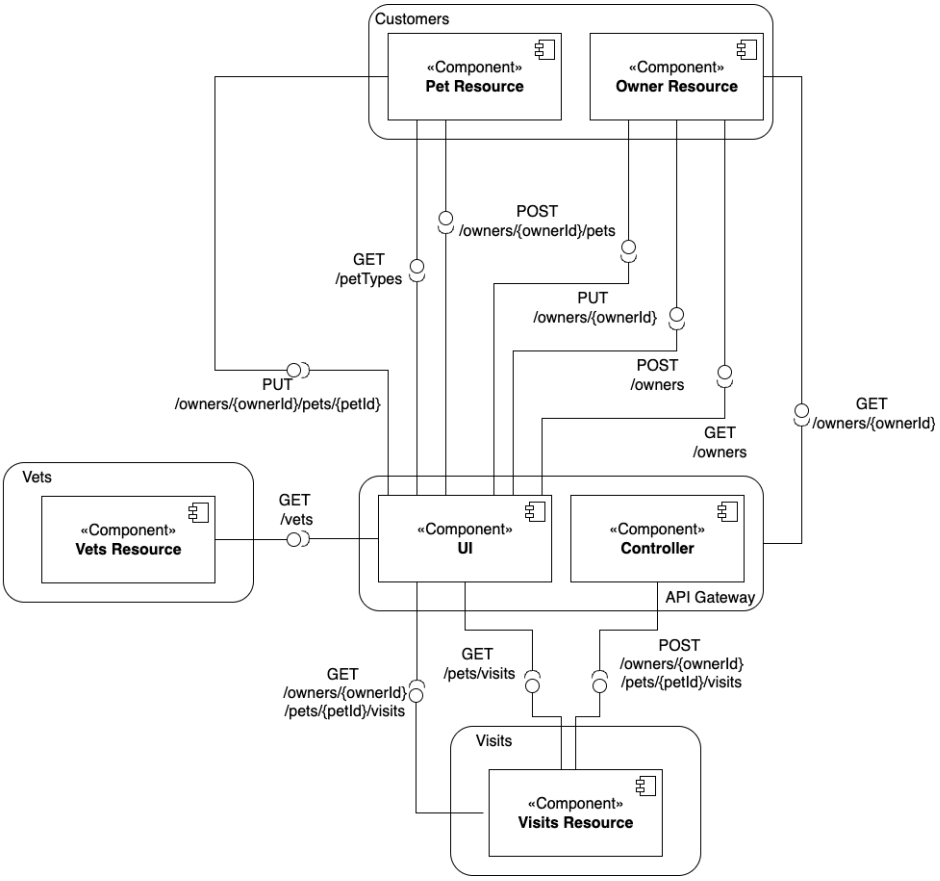

Figure 5.1: Interactions between services

- protoc-gen-go-grpc - Plugin of Protocol Buffer Compiler to generate Go code from protobuf service definitions

Having the service interactions clearly understood and the necessary tools installed, the conditions for progressing with the migrations are met.

5.2 Definition of gRPC Contracts

Based on the identified interactions, the next phase of the migration involves defining the gRPC contracts, which clearly specify the operations and their corresponding input and output data.

5.2.1 Structure and Conventions

With the proto files being the source of truth, their organization and structure directly influence the readability and clarity of interfaces. With that goal in mind, conventions were defined, including aspects such as the file structure, consistent naming conventions for services and messages, and package structure.

For a general structure, every proto file follows a layout consisting of five different sections, organized in a specific order:

1. Syntax declaration - Specifies the Protocol Buffers version

2. Package declaration - Specifies the logical namespace
3. Language-specific options - Generation configuration options for the target programming languages
4. Import statements - Dependencies of other external proto files or other libraries
5. Service/Message definitions - Core contracts and data structures

When every proto file follows this structure, it facilitates code readability and ensures that teams can quickly navigate and understand any file. Specifically, the package declaration should be (sub-domain is only included if applicable):

organization.domain.sub-domain.service

In addition to the file structure and package naming, it was also defined naming conventions to be applied transversally to the elements of the gRPC contracts. Table 5.1 presents the main naming rules adopted for services, methods, messages and endpoints, followed by some examples.

Element Type	**Convention**	**Example**
Services	PascalCase + "Service" suffix	OwnerService
Methods	PascalCase verbs	CreateOwner, GetOwner
Request Messages	PascalCase + "Request" suffix	CreateOwnerRequest, GetOwnerRequest
Response Messages	PascalCase + "Response" suffix	CreateOwnerResponse, GetOwnerResponse
HTTP Endpoint	Lowercase pluralized nourns	/owners

Table 5.1: Contract elements naming conventions

The definition and enforcement of these conventions and structures contributed to the creation of clearer contracts, making them easier to interpret. It was especially beneficial during the migration by enabling different services to share the same consistent patterns, and, consequently, facilitating their integration.

5.2.2 Contracts Organization and Management

In the migration to gRPC, the central management of the API contracts is highly relevant. For this reason, it was decided to create a new repository, completely separate and exclusively dedicated to storing proto files, where the services and messages are defined by all systems in the organization. The main reason behind this decision is the promotion of consistency between APIs and teams, version control and, more importantly, keeping track of what has already been migrated.

This approach facilitates the application of transversal rules of organization, naming and validation. For example, organizations that may want to invest in automatic mechanisms to detect inconsistencies, duplication or circular dependencies between definitions are able to do so with a centralized repository for proto files. Not only that, but having a single access point to all interface definitions makes the visibility among teams easier, especially in a context with a large number of interdependent services. This repository also serves as an updated catalog to easily identify which services have already been migrated to gRPC, or at least are available to be called through gRPC.

Another key advantage is the decoupling between the contracts and the services that expose them. By keeping the proto files outside the repository of the services, it becomes possible to maintain, evolve and version the contracts independently (as long as compatibility is ensured). That separation is also beneficial for automation processes, as it makes it much easier the creation of dedicated pipelines to compile the contracts and publish the generated stubs as artifacts in registries in different programming languages.

However, as with any decision, there are trade-offs: this approach also needs to deal with some challenges. The synchronization between the contracts and the services that expose/-consume them requires attention, especially in environments where changes are constantly being made. The absence of automatic validation between the service's code and the central contracts can cause inconsistencies if CI practices are not adopted. That is why CI plays a crucial role when maintaining contracts between services (Bogner, Fritzsch, and S. Wagner 2021). Additionally, the process of making changes in a contract can become more bureaucratic, requiring more coordination between teams.

As an alternative, an approach where each service keeps its own proto files inside the project could be adopted. The truth is that it is a much simpler approach as it simplifies local development and ensures direct alignment between the contract and the implementation. However, it makes it much more difficult to share between services or teams, especially services with a large number of clients, and reduces the capacity of supervising the adoption of the structures and conventions defined.

Although the chosen project does not have very high complexity, for the sake of the migration and in order to cover more scenarios, the approach of having a centralized and separated repository has shown to have clear advantages over the local storage of the proto files.

Inside the repository, the folder structure should be similar to the package name in the gRPC file: the top level should be the domain, followed by the subdomain (when applicable) and then the service (Amaral 2025b). Figure 5.2 shows an example of the folder structure inside a domain called "Petclinic".

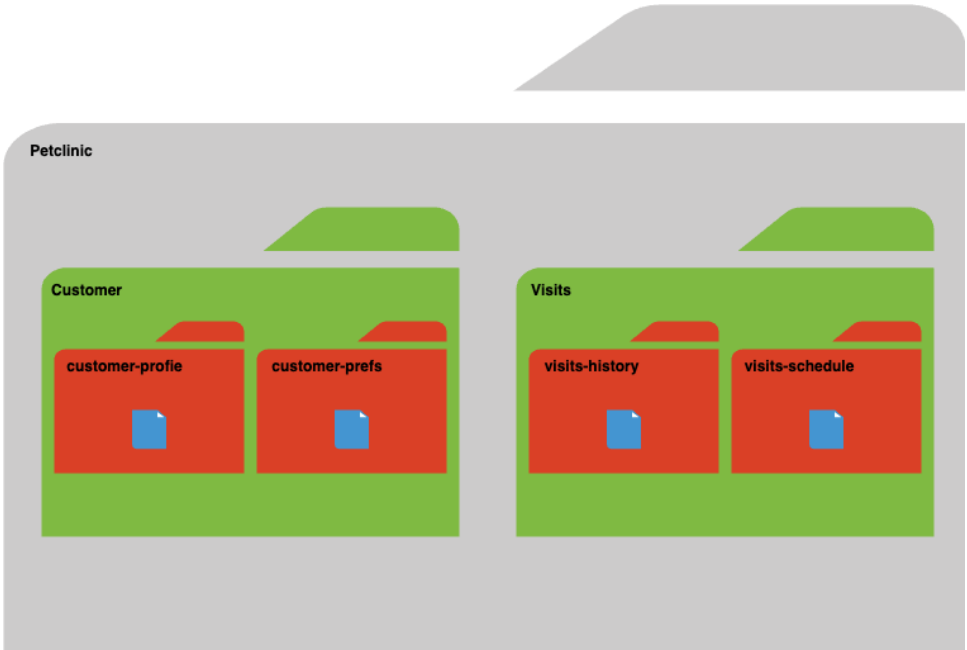

Figure 5.2: Example of the folder organization inside the proto repository

Each service should then have two different types for proto files (Amaral 2025b): one called "Types.proto" where all the messages (both requests and responses) are defined; and another for service definitions. About the last type, each service should have as many of these files as there are resources. For example, if a REST API has a resource called "Owners" and another called "Pets", there should be three files inside the service folder: "Type.proto", "Owner.proto", and "Pet.proto".

This way, it becomes even easier to find the needed definitions, whether they are messages or services.

5.2.3 Creation of Contracts

The creation of contracts consists of the definition of services and messages that compose the gRPC contracts, based on the operations and data obtained from analysing the project.

Specifically for the messages, the contract creation was done with the help of tools based on Artificial Intelligence (AI). The use of these tools aimed to accelerate the process of writing the contracts and gather initial ideas for the message structure. However, the results were thoroughly reviewed, adjusted to the domain and no private data was shared with any agent (Tech 2024).

After minor refinements, the prompt used returned fairly accurate responses. The prompt used for each class representing the message was:

POJO pasted here

Based on this POJO, can you create a protobuf message definition? Please, use the proto3 syntax and the package should be petclinic.customer.

The agent used at the time of the prompt was GPT 4.1. The output responses were considerably good, as all Java data types were correctly mapped to Protobuf data types and the attributes and relationships between objects were correctly generated. The output only required minor changes to match the conventions defined in section 5.2.1, for example adding the "Request" and "Response". The listing 5.1 shows one of the message definitions generated by the AI agent.

```
syntax = "proto3";

package petclinic.customer;

import "petclinic/customer/pet.proto"; // Assuming Pet is defined here

option java_multiple_files = true;
option java_package = "petclinic.customer";
option java_outer_classname = "OwnerProto";

message Owner {
  int32 id = 1;
  string first_name = 2;
  string last_name = 3;
  string address = 4;
  string city = 5;
  string telephone = 6;
  repeated Pet pets = 7;
}
```

Listing 5.1: Message definition gererated by AI

The message definitions generated by the AI agent successfully mapped the fields of the POJO to the corresponding Protocol Buffers types, including, in the shown case, the repeated field for pets. About the pet property (and when there were nested objects), with previous prompts, the agent attempted to define the message types for nested objects with suggested properties. However, after the refinement, the responses did not include any nested message definitions, which is the desired behaviour.

The changes made to the generated messages were only related to the language-specific options. That was the case with the "java_package" options, which were changed to match the structure of the Spring Petclinic Microservices project, and the "java_outer_classname" option, which was changed to match the service name with the suffix "TypesProto". Additionally, a property called "go_package", which is used by the gRPC Gateway to generate the Go code, was added. The listing 5.2 partially presents the final version of the message definitions for the Customer service.

```
syntax = "proto3";

package petclinic.customers;

option go_package = "petclinic/grpcgateway/gen/proto/customer";
option java_package = "org.springframework.samples.petclinic
.customers.grpc.gen.customer.types";
option java_multiple_files = true;
option java_outer_classname = "CustomerTypesProto";

message Pet {
  int32 id = 1;
  string name = 2;
  string birth_date = 3;
  PetType type = 4;
  Owner owner = 5;
}

message PetType {
  int32 id = 1;
  string name = 2;
}

message Owner {
  int32 id = 1;
  string first_name = 2;
  string last_name = 3;
  string address = 4;
  string city = 5;
  string telephone = 6;
  repeated Pet pets = 7;
}

message CreateOwnerResponse {
  Owner owner = 1;
  bool created = 2;
}

message GetOwnerResponse {
  Owner owner = 1;
  bool found = 2;
}

// Other message definitions...
```

Listing 5.2: Partial message definitions for Customer service

Regarding the service definitions, the process was slightly different. As the generation by the AI agent required more context, such as knowing the types of input and output data that had been converted to and the endpoints for the gRPC-Gateway (as seen later), the generation option was discarded. Instead, it was done manually. It did not require a large amount of effort since all the operations had already been identified. The service definitions were created in separate proto files, following the conventions defined in the preceding sections. The full contract definitions for customer service can be found in Appendix B.

5.3 Code Generation

Now that contracts are defined, the next step is to generate the code for the server, the stub and the gateway. This automatic generation is a key advantage of gRPC, since it allows the creation of client and server code in several programming languages without any manual effort from the developer. This not only reduces the risk of human error during contract implementation but also ensures that the code is consistent with the defined contracts.

For the sake of the migration, the process was done locally; however, once the needed commands and plugins are identified, it can be easily automated in a CI pipeline. Furthermore, it is highly recommended to automate the generation of code in a CI pipeline, as it ensures that the generated code is always up-to-date with the contracts and reduces the risk of inconsistencies between the contracts and the specific implementation.

The main difficulty in implementing it on a CI pipeline is ensuring that it can generate code for multiple languages, since gRPC contracts can be used in different programming languages, while supporting the language-specific options in the proto file. This requires keeping the proto files clean while, at the same time, applying language-specific configurations like "java_package" or "go_package", among others. A scalable solution to this problem is to externalize those options, using tools like BUF (Build 2025) or scripts, to insert them during the code generation process.

The code generation was two different moments: the first one was to generate the server and stub code, which is done in Java, and the second one was to generate the gRPC Gateway code, which is done in Go [1]. The root folder of both the Spring Petclinic Microservices project and the proto files repository was placed in the same directory.

To generate the server code, in Java, it was navigated to the target project inside the Spring Petclinic Microservices project and the command was executed from there. To successfully run the command, the protoc and the Protoc Gen gRPC Java plugin were needed. List 5.3 shows the used command.

```
protoc
 -I ../../spring-petclinic-proto
 -I ../../googleapis
 --go_out=gen/proto
 --go_opt=paths=source_relative
 --go-grpc_out=gen/proto
 --go-grpc_opt=paths=source_relative
 --grpc-gateway_out=gen/proto
 --grpc-gateway_opt=paths=source_relative
   ../../spring-petclinic-proto/petclinic/customer/*.proto
```

Listing 5.3: Command to generate the server code

As specified in the proto files, the generated code was placed in the "src/main/java" folder, following the package structure defined in the proto files. To keep the code organized, the package structure chosen was "grpc/gen/{service-name}/types" for the types file and "grpc/gen/{service-name}" for the service definition files, where "service-name" is the name of the service being generated. The service implementations should then be inside the "grpc" package.

[1] At this point the project containing the Gateway code was already created but is explained more in depth later

For the Gateway code, in Go, the process was very similar. The command was executed from the root folder of the gRPC Gateway project, which is a project created inside the Spring Petclinic Microservices. The command just had a few additional options. Listing 5.4 shows the command.

```
protoc
 -I ../../spring-petclinic-proto
 -I ../../googleapis
 --go_out=gen/proto
 --go_opt=paths=source_relative
 --go-grpc_out=gen/proto
 --go-grpc_opt=paths=source_relative
 --grpc-gateway_out=gen/proto
 --grpc-gateway_opt=paths=source_relative
 ../../spring-petclinic-proto/petclinic/customer/*.proto
```

Listing 5.4: Command to generate the gateway code

The generated code files were placed in the "gen/proto" folder (Amaral 2025a), and then in the module specified in the proto file.

5.4 gRPC Gateway

It is common that a RESTful service has multiple clients. This may present a challenge when migrating to gRPC, as those clients often don't transit to gRPC immediately. For that reason, it is essential to adopt a strategy that ensures the compatibility between the new gRPC service and the existing REST clients. That scenario led to the adoption of the gRPC Gateway (gRPC-Gateway 2023), which was created exactly to solve this problem.

The gRPC Gateway is a plugin for the protoc that acts as an intermediary between the REST clients and the gRPC services. Figure 5.3 (Gateway 2025) shows the architecture of the gRPC Gateway.

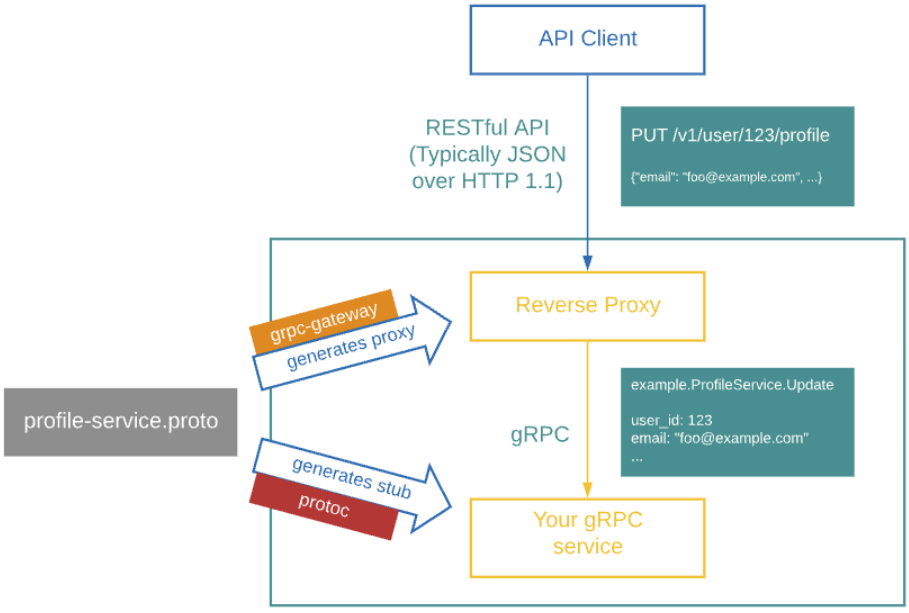

Figure 5.3: gRPC Gateway architecture

The way it works is it reads protobuf service definitions and generates a reverse-proxy server that translates REST HTTP API requests into gRPC calls. This allows existing REST clients to continue working without any changes, while the new gRPC service can be implemented and used by new clients.

The problem with the gRPC Gateway is that it depends entirely on the Go ecosystem. It is written in Go and the generated code is also in Go. This means that the generated HTTP server is written in Go and uses libraries and a runtime environment from the Go ecosystem. This presents a challenge, as a large number of projects, including the Spring Petclinic Microservices, are written in other languages, such as Java.

To overcome this problem, the decision was made to completely separate the responsibility of handling HTTP requests in an independent service developed in Go and using the code generated by the contracts created. It means that a new project was created inside the Spring Petclinic Microservices project, but it is totally independent of the other services (Amaral 2025a). Figure 5.4 shows the flow of a REST request until it reaches the service.

Figure 5.4: REST request flow with the gRPC Gateway

Ultimately, the gRPC Gateway exposes an HTTP server based on the annotations in the proto contracts, which define the mapping between gRPC methods and REST routes. Then, it translates those requests into gRPC request to the corresponding service and calls it as a client. The client response then follows the opposite flow: the gRPC Gateway receives the response from the service, transforms it into an HTTP response and sends it back to the original client.

5.5 Adding Tests

As seen in section 4.3.2, the Spring Petclinic Microservices project has a very low test coverage, with some services not having any tests at all. That scenario make is difficult to ensure that the changes in the communication protocol does not introduce any regressions.

To mitigate this problem, the decision was to add tests to the services that would be migrated to gRPC. To make the migration process easier, the tests created, mainly on the presentation layer, were designed in a way that they could be easily adapted to the gRPC services to be implemented. The input data should only have minimal changes (for example, changing from the REST request body to the gRPC request message) and only the calls to the service should be changed.

In the Customer service, the coverage was increased from 41% to 90% (Amaral 2025a), with only the main method not being tested (it did not contain any logic or behavior).

5.6 Server Migration

The server migration consists of implementing the gRPC services based on the previously defined contracts. The objective is to only change the communication protocol - replace

REST controllers with gRPC services. It can be done more securely now that the tests cover most of the application. As stated, each resource has been mapped to a gRPC service, and each operation has been mapped to a method in the service (Amaral 2025a). This means that the project will have as many gRPC services as there are resources in the REST API. The resources should be migrated one at a time.

At first, a new class for the gRPC service was created, extending the gRPC service base class. The dependencies present in the controller were injected into the gRPC service, so that the logic on each method could remain the same. The methods were then overwritten and implemented, calling the same dependencies as before and following the same logic. The only difference is that the input and output data should be changed to the gRPC requests and responses, respectively (Amaral 2025a).

The only major change in the logic is how the input data is mapped to domain entities and how domain entities are mapped to the output data. In the case of the Spring Petclinic Microservices, the input data is mapped to the domain entities (and the other way around) using mapper classes. The logic of those mapper classes needed to be adapted to accept the gRPC requests as input, when mapping to domain entities, and return the gRPC responses as output, when mapping from domain entities.

In scenarios where such mapping does not exist, meaning that the input data is directly used in the business logic, the situation is trickier. That happens because the classes containing the methods' behavior depend on and are tightly coupled to the structure of the input data. The recommendation here is that instead of changing those methods to use the gRPC requests, they should be refactored to work with domain entities. In fact, it is widely accepted that the inner layers of an application should not work with external representations of data (Martin 2009), such as Data Transfer Object (DTO) or gRPC requests. Fortunately, the Spring Petclinic Microservices project already follows this principle, so the methods were already working with domain entities.

On the other hand, on some endpoints, the project directly exposed some domain entities as a response to the requests. It slightly complicates the migration since gRPC requires that output messages are explicitly defined in the proto files. In those cases, additional mapping methods were created to convert the domain entities to the corresponding gRPC response messages (Amaral 2025a). This approach improves the overall quality of the project by avoiding the exposure of the internal data structures and reducing the coupling between the service layer and the presentation layer. Furthermore, it allows better control over the response format and defines more clear and stable structures for information exchange.

Once all those changes were made, the automated tests were obviously failing. Hence, the next step is to adapt the tests to the new gRPC services. Since the tests were already designed to be easily adapted, only minor changes to the input data and the calls to the service were required. The hardest part was to manually create the gRPC server to receive the requests from the tests. After those changes, the tests were run and the results were checked. If any test failed, it was necessary to check the implementation of the service. Some errors were caught by the tests, such as wrong error messages when dealing with invalid input data, highlighting the importance of having regression tests.

For further validation, Postman was used to manually test the gRPC service and the gRPC Gateway. For that, three different requests were made:

1. A request to an endpoint of the original, unchanged Petclinic project.

2. A request directly to the gRPC service.
3. A request to the gRPC Gateway.

The output of the first request serves as the source of truth for the next comparisons, ensuring that the migration would not change any behaviour or data returned by the application. Once the correct output was obtained, the tests focused on the gRPC service using Postman's gRPC tool. The main goal of this test was to verify if the gRPC contracts were correctly implemented and that the returned value matched the one on the original API call. Finally, the same scenario was tested, but calling the gRPC Gateway instead. In this case, a regular REST call was made to the gateway, which was then successfully translated to a gRPC call, showing the gateway's capabilities of allowing the simultaneous existence of REST and gRPC services, and then returned the exact same response.

The results obtained in each test demonstrated that the behaviour of the system was consistent, no matter the technology used, and ensured that migration did not introduce any incompatibilities. So far, the clients have kept consuming data via REST calls exposed by the gateway.

5.7 gRPC Stubs Distribution

With the server fully migrated, the gRPC Gateway working correctly and the clients consuming from it, the clients should be migrated now. However, to avoid the local generation after generation of code, a strategy of centralized distribution of gRPC stubs was adopted. That strategy involves packaging the classes generated by proto files in a dedicated Java Archive (JAR) artifact. Later, that artifact is imported and used by the clients as a dependency. Conceptually, this strategy replicates the generation of stubs through a CI pipeline. As explained in section 5.3, it is recommended that the code generation is automated, referring to a CI pipeline: a single producer publishes the stubs and multiple clients use a stable version of the published artifact.

From a migration point of view, this approach reveals key advantages. First, it ensures consistency among clients as all share the exact same code for a specific version of the contracts. Secondly, it reduces the operational complexity of the migration by taking away the need for each client to configure the protoc and the necessary plugins and generate the code once again. Instead, they entirely depend on a conventional artifact (a JAR in this case). And finally, it decreases the migration time needed since the clients only need to incorporate the artifact in order to start consuming.

There are, however, some drawbacks to this approach. Mainly, it requires versioning of the contracts to keep their stability. A good adoption would be, for instance, Semantic Versioning 2.0.0 (Preston-Werner 2013). It divides a version into three different parts: MAJOR.MINOR.PATCH. Each part should be incremented according to the changes made to the contracts. Major increments should be done when there are breaking changes in the contracts. That should be the case, for example, of removing or renaming fields. Incrementing the Minor part should be done when there are backward-compatible changes, such as adding new services or messages. The Patch part is incremented when fixing backward-compatible bugs. So, an artifact containing the code for the Customer service would be, for example, spring-petclinic-customer:1.4.2.

While being an efficient solution in the current work context, it has limitations. Since it is a manual process, there is the risk of clients having outdated versions when compared to the

server. Furthermore, it only benefits Java clients - consumers with other languages need equivalent artifacts.

However, it can now be easily adapted in a pipeline. The proposed steps are:

1. Linting and breaking changes validation.
2. Stub generation for a set of languages (can be provided by input).
3. Build the necessary artifacts.
4. Publish the artifacts in a repository.
5. Notifying client teams (optional).

It is worth noting that the gRPC stubs distribution through artifacts does not conflict with the existence of a repository for storing the Protocol Buffer files. Rather, they complement each other. While the repository acts as a single source of truth regarding the contracts definition, ensuring their versioning and visibility, the artifacts distribution is a concrete instance of the contracts with the goal of making the integration easier for engineering teams.

5.8 Client Migration

The client migration was a fairly straightforward step since most of the effort had already been done. It simply consisted of importing the artifact with the stubs, creating a gRPC client, using the stubs to call the methods in the server, and making the necessary adjustments in the automated tests (Amaral 2025a). With the clients fully migrated, the gRPC Gateway can be decommissioned.

For the functional tests, once again, the original unchanged project was run at the same time with the fully migrated project. Since both produced the same result when performing the same operations, it can be safely said that the migration was functionally successful.

Chapter 6

Discussion of results and Guidelines

The migration process described in the preceding chapter not only allowed to verify the truthfulness of the concepts identified in the literature review, but also check in a practical context the challenges faced in a transition from REST to gRPC. The current chapter describes the results obtained through the migration process, analyzing how it fulfilled the requirements of a migration and compares the developed process with the recommendations from the literature and industry cases, evaluating how they are (or are not) aligned and reinforcing the validity of the lessons learned. Furthermore, the discussion also reveals the challenges faced, limitations or room for improvements.

6.1 Achievement of Migration Objectives

The migration from REST to gRPC implemented in this work aimed to validate the approaches identified through the literature review while providing practical insights for engineering teams. The objectives established were successfully achieved through a systematic approach that combined theoretical research with practical implementation.

The review of academic literature and industry cases provided the base knowledge necessary for understanding existing migration methodologies. Despite the limited number of academic sources available (only two relevant studies identified), the analysis of industry experiences from companies such as WePay, Google Cloud, and LinkedIn effectively complemented the research, revealing common patterns and validating existing strategies.

The migration of the Spring PetClinic Microservices project from REST to gRPC demonstrated the applicability of the identified methodologies in a practical context. The project served as an adequate test case, allowing for the validation of key concepts, including contract-first development, gradual migration with backward compatibility, and automated code generation.

Through the practical implementation, several best practices were validated and new insights were discovered, contributing to a deeper understanding of REST to gRPC migration processes. The experience provided concrete evidence for the effectiveness of strategies identified in the literature while revealing additional considerations specific to Java-based microservices architectures.

The success of the migration can be evaluated against the objectives defined to determine its success. In first place, the migrated system demonstrated complete functional equivalence with the original REST implementation. The tests using three validation approaches - original REST endpoints, direct gRPC calls, and mediated requests through gRPC Gateway

- showed that all operations produced the same results. This validates that the migration preserved all business logic and data integrity.

Besides the functional equivalence, a key indicator of success is the stability of the system during the migration. The coexistence of REST and gRPC, allowed by the gRPC Gateway, ensured that there were no service disruptions by maintaining backward compatibility with existing REST clients while enabling new gRPC clients to consume the services directly. This is an essential aspect to decrease risks and certify a gradual and controlled adoption. The centralized distribution of gRPC stubs through JAR artifacts simplified the integration process for client applications even more. By eliminating the need for individual teams to manage Protocol Buffer compilation and code generation, the approach reduced operational complexity and accelerated adoption, especially beneficial in corporate environments with multiple development teams. From an automation point of view, the process identified commands, plugins and configurations necessary to create conditions for its automation.

It is also important to note that the migration does not provoke any downtime, other than a regular release, and combines different languages, mainly Java and Go. This experience shows that the migration is viable even in distinct environments as long as they are supported by gRPC. That means that the proposed guidelines can be adopted in other technological contexts without losing their applicability.

Regarding the information obtained during the literature review process, it is important to show how it was incorporated. Such knowledge provided a solid base of concepts but also practical guidelines and influenced the decisions made during the process. The decisions were:

- **Iterative and gradual process:** Inspired by the recommendations of the literature and in all industry cases, the migration process was iterative and gradual, allowing the simultaneous existence of REST and gRPC. Additionally, proper planning was made prior to the migration.
- **Identification and documentation of endpoints:** According to the literature, one of the first and most important steps in a migration is mapping all the entry points in the API and understanding the exchange of data. A diagram was designed showing the interaction between services.
- **Contract centralization:** Following the example of WePay, the gRPC contracts were all centralized in a repository for the effect.
- **Automate everything possible:** One of the main lessons is to maximize automations, since identification of endpoints to validations. Unfortunately, it was not possible to automate some parts of the migration. However, gRPC messages were generated with the help of AI agents and code generation was prepared to be automated in CI pipelines.
- **Usage of support tools:** Literature suggested the usage of tools to support endpoint identification and tests. Postman was used to compare and confirm the responses of the services. Additionally, industry cases highlighted the usage of bridge tools, like gRPC Gateway. For being open-source, gRPC was also used, reflecting the strategies of coexistence of REST and gRPC.

- **Tests and validation:** The importance of regression tests, mentioned in the literature, was applied by reforcing the test coverage before the migration, ensuring no errors were introduced and changes in the behaviour.
- **Strategies inspired by other migration types:** Literature recommends adopting strategies common in other migration types, such as phased rollout and regression, to decrease risks and guarantee stability. Such practices were followed in the planning and execution of the migration.

As mentioned in the section 2.1, regarding the migration types, the developed process can be categorized into two different types: wrapping and incremental. Wrapping because, while the migration was not concluded, gRPC Gateway was used as a translation layer between REST and gRPC. That means that part of the system was wrapped to be accessible as a gRPC service without the need to change any code in both server and clients. On the other hand, it can also fit in the incremental migration type. Mainly, because the migration was gradual, keeping compatibility between REST and gRPC during the process and was made in increments: first, the servers, migrating one endpoint at a time, and then the clients, migrating one consumer at a time. This is typical of an incremental migration, where there's a phased substitution and keeping the system operational.

That being said, the performed migration not only achieved the defined goals but also showed that the recommendations by the literature and industry cases are effective. Figure 6.1 shows a diagram with the final solution.

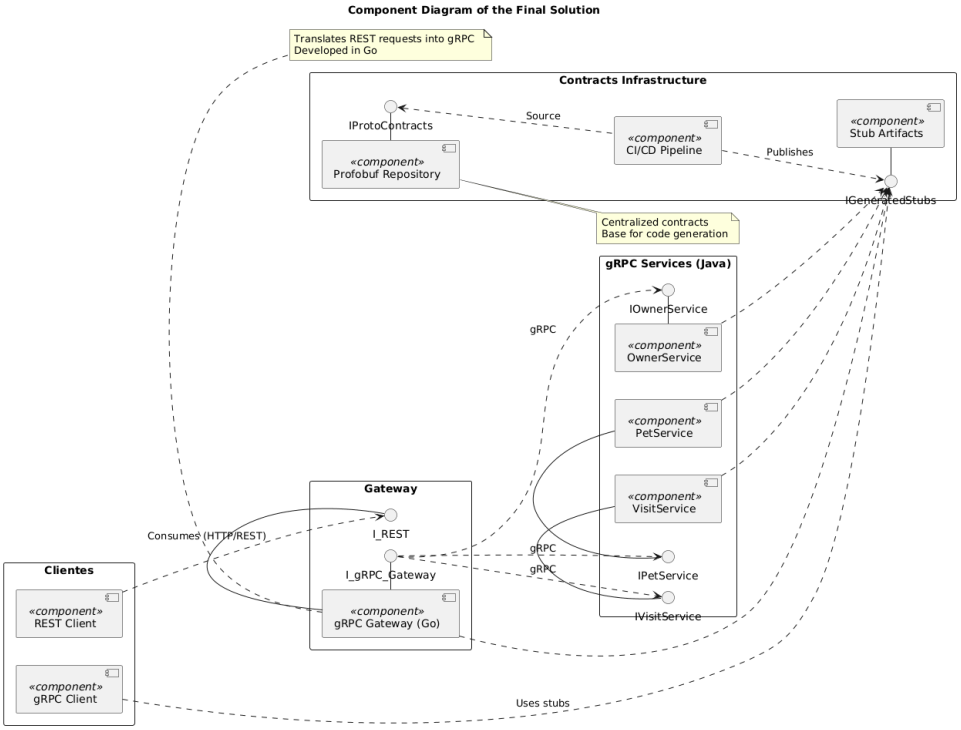

Figure 6.1: Component Diagram of the Final Solution

6.2 Challenges and Improvements

Although the requirements were fulfilled, the process was not immune to challenges that required adjustments in the implementation. The analysis of these challenges is important, as it also provides an opportunity to identify possible improvements to consider in future migrations.

In the initial stages of the migration, the absence of a formal API Specification led to manual analysis of the code to identify endpoints and data structures, which delayed the migration and increased the process's vulnerability to errors. This was made even more serious by the lack of or low coverage of tests. While the project gaps, another challenge was the direct exposure of domain entities to the external world, which once again needed a refactor. Specifically, methods needed to be adapted to work with gRPC messages instead of domain entities.

Another unexpected challenge is related to the gRPC Gateway. It introduced additional complexity as it was managed in Go. For that reason, it was necessary to learn the syntax of the language as well as change the gRPC contracts to generate the necessary code for the gateway.

However, the biggest challenge was the strategy for gRPC stubs distribution. In order to keep the migration as simple as possible, redundant efforts needed to be avoided. In a scenario without a strategy for stubs distribution or where each team managed its own stubs, the migration could be delayed. Mainly because the coordination between the client and server depended on a manual process prone to error, which needed to be learn by all teams, rather than an automated flow. In the end, it was not possible to fully automate it, but the foundation for making that transition was laid.

And that is precisely where the main improvement resides. The generation and publishing of stubs for all necessary languages should be on CI pipelines instead of local builds. To make it even better, linting, automatic detection of errors or breaking changes should be included. Additionally, another mechanism can be included in the build of the projects to check if the services implementation is aligned with the centralized contracts.

Another improvement would be the adoption of observability mechanisms during and after the migration in order to quickly identify problems or improvements.

6.3 Guidelines

Based on the successful implementation of a migration from a REST system to gRPC, it was possible to identify a set of guidelines to support engineering teams in planning and/or executing the migration. Those guidelines should not be seen as a strict step-by-step formula, but rather as recommendations that, when adapted to an organizational context, can provide a safer and efficient transition.

6.3.1 Initial Preparation

One key aspect identified that ensured the efficiency of the migration is the careful preparation of the environment and understanding of the current system. Before executing the migration, it is essential to map all the endpoints and their respective interactions, if it does not already exist. That way, it is easier to keep track of what has already been migrated

and what remains to be migrated. If all clients are not known to a certain server, at least the endpoint and its input/output data should be well-documented. That avoids a manual analysis of the code, which can significantly delay the migration, especially when the team is not entirely familiar with the project. On the other hand, it makes the definition of gRPC contracts much easier and straightforward. While it may not be clear in the initial stages, the prior installation of the necessary tools and plugins, such as the Protocol Buffer compiler and its plugins, as well as testing tools, can smooth the process and avoid possible interruptions.

6.3.2 gRPC Contracts Management

Regarding the gRPC contracts management, the adoption of conventions for the structure and naming of the files is important. For example, in the case of the implemented migration, the usage of Pascal Case was opted for, along with the suffixes Service, Request, and Response. Such conventions improve the legibility, maintenance and efficient integration between teams. Each organization should follow its own conventions and not be bound to the ones presented.

Another valuable approach is to completely separate the contracts in an independent repository, apart from the application's repository. It allows for better consistency, versioning, and visibility across teams. It is also easier to integrate with automations. To make it even more organized, the repository was structured following the hierarchy of domain/subdomain/service, which simplifies the location of the files. Once again, each organization should enforce its own hierarchy as long as it fits its needs.

Additionally, two types of files are created: types.proto, which contain the message structure and other files for the service definition. Finally, it was included in the language-specific options in the contracts; however, the externalization of these options likely benefits automations and code generation in multi-language environments.

6.3.3 Code Generation and Automation

For code generation and automation, the importance of generating, publishing, and distributing artifacts in CI pipelines is highlighted so that the generated stubs are always aligned with the contracts. To complement this, automatic validations can also be implemented, such as applying lint rules or detecting breaking changes. The extra mile regarding this topic is checking whether the service implementation is, in fact, following the contract. With both these mechanisms, the consistency between contracts and stubs, and contracts and implementation is fully ensured. Once the stubs are generated, they should also be made available using a systematic versioning system, for example, Semantic Versioning 2.0.0. It allows stricter control over compatibility and a predictable evolution of the contracts over time.

Optionally, when a new contract version is published, a notification is sent to the interested teams, so that they can start planning the transition to the newer version.

It is also important to note that teams may leverage AI to automate some specific tasks, always checking the correctness of the outputs.

6.3.4 REST and gRPC Coexistence

To avoid disruptions in the applications, it is important to temporarily allow the simultaneous existence of REST and gRPC services. Hence, there should be a layer to make it possible, allowing compatibility with non-migrated clients or a gradual migration. In the developed migration that was achieved with the gRPC Gateway developed in Go. The gateway should not contain any logic. Rather, it should only be seen as an adapter layer.

However, that is not enough. In order to keep functional equivalence, it is important that each REST resource is mapped to a gRPC service and that the gRPC service keeps the same property names as in the original REST requests (in and out). That way, no changes in clients are needed - they keep consuming REST as if nothing had changed. Once the clients are migrated, the gateway can be easily decommissioned.

6.3.5 Tests and Validation

Regarding tests and validation, the experience during the migration showed the importance of having good test coverage before starting the process, mainly on regression tests. This measure ensured that changes introduced by the protocol change did not provoke any unwanted behavior. Furthermore, the automated tests in the presentation layer should be adapted so that they can be easily changed when the code migration to support gRPC is complete. Finally, the system as a whole should be tested by making requests to all the layers, the gRPC service and the gateway, and the original system. It validates the functional success of the migration.

6.3.6 Assessment of Migration's Success

The objectives established in section 4.2 guide the assessment of the migration's success. For the migration to be successful, the criteria to be met is:

- **Simplicity:** The process should be the least complex possible, keeping compatibility with non-migrated clients. Furthermore, to evaluate the particular criteria, the ease with which development teams can consume and evolve the new gRPC services can also be assessed.
- **Availability:** The system's downtime should not exceed the downtime of a regular release. It is equally important that the migration is planned and performed in a way to minimize the impact on end users to ensure that the whole process is smooth and limited to strictly necessary interruptions.
- **Functionalities:** All existing functionalities prior to the migration must be preserved.
- **Performance:** An improvement in performance is expected after the migration, aligning with the advantages of gRPC over REST.

Chapter 7

Conclusion

In conclusion, the goal of the work is to provide guidelines and good practices for engineering teams considering a migration from a REST system to gRPC, as gRPC may bring many benefits to software applications. That was achieved by studying various types of migrations, understanding how gRPC and other RPC frameworks work and how gRPC compares to REST, exploring how the literature and industry handle those migrations and then implementing a migration in a project. The chapter presents the achievements, challenges and future work.

7.1 Achievements and Challenges

The process was able to verify that all the recommendations, both in the literature and industry, are reliable while consolidating a set of guidelines that can be applied to other projects. The results demonstrate that it is definitely possible to replace a REST-based system with gRPC and preserve functional equivalence. Through the structured definition of contracts, the automatic generation of code and the usage of a gRPC Gateway it was shown that the transition can be made gradually and minimize the risks of interruption. Additionally, the centralization of gRPC contracts in a repository alongside the automatic distribution of the generated stubs contributed to better consistency and alignment when there are several teams involved.

Challenges were faced, not only during the migration, but also during the investigation. One of them is the lack of academic peer-reviewed sources regarding the subject under study, which made the understanding of current practices in a migration from REST to gRPC scenario more complex. Due to the number of requirements, it was also difficult to find a suitable project to conduct the migration. Finding an open-source system, licensed, with documentation and that allowed the application of migration strategies represented a significant challenge. Many available projects were either too simple, not accurately reflecting the complexity of real-world scenarios, or too complex, making it difficult to focus on the migration process. That step required a careful analysis to ensure that the chosen project provided relevant results to the work.

In the end, the results verify that a migration can be successfully performed as long as it has a proper plan and tools, good practices of automation and testing and mechanisms of compatibility.

7.2 Future Work

For future work, the automation can take a step forward by automating the whole process. A static analysis of code (controllers, routers, DTOs, validations, annotations, etc) can identify the endpoint present in a REST API, its methods, input and output data, and then create the gRPC contracts from it. With such information, it would also be possible to generate the code for the gRPC Gateway.

In a less ambitious alternative, an automation to create gRPC contracts from API Specification files. The migration impact, when compared to REST, in performance and scalability could also be evaluated.

Bibliography

Abernethy, Randy (Mar. 2019). *Programmer's Guide to Apache Thrift*. Manning Publications.

Almonaies, Asil, James Cordy, and Thomas Dean (Jan. 2010). "Legacy system evolution towards service-oriented architecture". In: *Queen's University*.

Althani, Bashair and Souheil Khaddaj (Oct. 2017). "Systematic Review of Legacy System Migration". In: *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*.

Amaral, Hugo (July 2025a). *Spring Petclinic Microservices gRPC*. url: `https://github.com/amaralhugo30/spring-petclinic-microservices-grpc`.

– (July 2025b). *Spring Petclinic Microservices gRPC Proto*. url: `https://github.com/amaralhugo30/spring-petclinic-proto`.

Apache (Jan. 2023). *About Apache Thrift*. url: `https://thrift.apache.org/about`.

Belshe, Mike, Roberto Peon, and Martin Thomson (May 2015). *Hypertext Transfer Protocol Version 2 (HTTP/2)*. url: `https://datatracker.ietf.org/doc/html/rfc7540`.

Birrell, Andrew D. and Bruce Jay Nelson (Feb. 1984). "Implementing remote procedure calls". In: *ACM Trans. Comput. Syst.*

Bisbal, Jesus et al. (Jan. 1998). "An Overview of Legacy Information System Migration". In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*.

Bogner, Justus, Jonas Fritzsch, and Stefan Wagner (June 2021). "Industry practices and challenges for the evolvability assurance of microservices". In: *Empirical Software Engineering*.

Brodie, M.L. and M. Stonebraker (1995). *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan Kaufmann Publishers.

Build, BUF (Jan. 2025). *BUF*. url: `https://buf.build/`.

CASP (Jan. 2024). *Critical Appraisal Skills Programme*. url: `https://casp-uk.net/casp-checklists/CASP-checklist-qualitative-2024.pdf`.

Community, Spring (Apr. 2025a). *Spring Petclinic Microservices*. url: `https://github.com/spring-petclinic/spring-petclinic-microservices`.

– (June 2025b). *The Spring PetClinic*. url: `https://spring-petclinic.github.io/`.

Dineen, T.H. et al. (Feb. 1988). "The network computing architecture and system: an environment for developing distributed applications". In: *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, pp. 296–299.

Fielding, Roy Thomas (Dec. 2000). "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis.

Fowler, Martin (Aug. 2014). *Microservices and the First Law of Distributed Objects*. url: `https://martinfowler.com/articles/distributed-objects-microservices.html`.

Gateway, gRPC (Aug. 2025). *gRPC Gateway*. url: `https://github.com/grpc-ecosystem/grpc-gateway`.

gRPC (Dec. 2023a). *About gRPC*. url: `https://grpc.io/about/`.

gRPC (Sept. 2023b). *gRPC Motivation and Design Principles*. url: https://grpc.io/blog/principles/.

– (Dec. 2023c). *Introduction to gRPC*. url: https://grpc.io/docs/what-is-grpc/introduction/.

gRPC-Gateway (Jan. 2023). *gRPC-Gateway*. url: https://grpc-ecosystem.github.io/grpc-gateway/.

Hemmati, Hadi (Aug. 2015). "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*.

IEEE (Jan. 2020a). *IEEE AUTHOR ETHICS GUIDELINES*. url: https://journals.ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Author-Ethics-Guidelines.pdf.

– (Jan. 2020b). *IEEE Code of Ethics*. url: https://www.ieee.org/about/corporate/governance/p7-8.

Indrasiri, Kasun and Danesh Kuruppu (Jan. 2020). *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media.

Indrasiri, Kasun and Prabath Siriwardena (Nov. 2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*. apress.

Kitchenham, Barbara and Stuart Charters (July 2007). "Guidelines for performing Systematic Literature Reviews in Software Engineering". In: *EBSE Technical Report*.

Kumar, Prajwal Kiran et al. (Sept. 2021). "Performance Characterization of Communication Protocols in Microservice Applications". In: *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*.

Lee, Yunhyeok and Yi Liu (Apr. 2022). "Using refactoring to migrate REST applications to gRPC". In: *Proceedings of the 2022 ACM Southeast Conference*, pp. 219–223.

Liskin, Olga, Leif Singer, and Kurt Schneider (Aug. 2012). "Welcome to the Real World: A Notation for Modeling REST Services". In: *IEEE Internet Computing*.

Mark Slee, Aditya Agarwal and Marc Kwiatkowski (Apr. 2007). "Thrift: Scalable Cross-Language Services Implementation". In: *Facebook*.

Martin, Robert (Sept. 2009). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. PEARSON EDUCATION.

Nielsen, Henrik et al. (June 1999). *Hypertext Transfer Protocol – HTTP/1.1*. url: https://www.w3.org/Protocols/rfc2616/rfc2616.html.

Pautasso, Cesare and Erik Wilde (Apr. 2010). "RESTful web services: principles, patterns, emerging technologies". In: *Association for Computing Machinery*, pp. 1359–1360.

POSTMAN (2023). *2023 State of the API Report*. url: https://www.postman.com/state-of-api/2023/api-global-growth/#api-global-growth.

Preston-Werner, Tom (Jan. 2013). *Semantic Versioning 2.0.0*. url: https://semver.org/spec/v2.0.0.html.

Rakowski, Krzysztof (Dec. 2015). *Learning Apache Thrift: Make applications cross-communicate using Apache Thrift!* Packt Publishing.

Ramgopal, Karthik and Min Chen (Oct. 2024). *gRPC Migration Automation at LinkedIn*. url: https://www.infoq.com/presentations/grpc-restli/.

Rotem-Gal-Oz, Arnon (Jan. 2008). *Fallacies of Distributed Computing Explained*. url: https://arnon.me/wp-content/uploads/Files/fallacies.pdf.

Santos, Nuno Martins (Oct. 2020). *Protobuf lab session*. url: https://farfetchtechblog.com/en/blog/post/protobuf-lab-session/.

Savolainen, Juha and Varvana Myllärniemi (Sept. 2009). "Layered Architecture Revisited – Comparison of Research and Practice". In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*.

Shah, Parantu K et al. (May 2003). "Information extraction from full text scientific articles: Where are the keywords?" In: *BMC Bioinformatics*.

Strebel, Daniel and Omid Tahouri (Aug. 2024). *From gRPC to RESTful APIs: Expose your gRPC services to the REST of the world*. url: `https://cloud.google.com/blog/products/api-management/bridge-the-gap-between-grpc-and-rest-http-apis`.

Tacconelli, Evelina (Apr. 2009). *Systematic reviews: CRD's guidance for undertaking reviews in Health Care*. University of York.

Tanenbaum, Andrew S. et al. (Dec. 1990). "Experiences with the Amoeba distributed operating system". In: *Association for Computing Machinery (ACM)*, pp. 46–63.

Tay, B. H. and A. L. Ananda (July 1990). "A survey of remote procedure calls". In: *SIGOPS Oper. Syst. Rev.*, pp. 68–79.

Tech, Georgia (Jan. 2024). *Effective and Responsible Use of AI in Research: Guidance for Performing Graduate Research and in Writing Dissertations, Theses, and Manuscripts for Publications*. url: `https://grad.gatech.edu/sites/default/files/documents/Guidance%5C%20for%5C%20Effective%5C%20and%5C%20Responsible%5C%20Use%5C%20of%5C%20AI%5C%20in%5C%20Research.pdf`.

Wagner, Christian (Jan. 2014). *Model-Driven Software Migration: A Methodology*. Springer Fachmedien Wiesbaden.

Weerasinghe, L.D.S.B and I Perera (Dec. 2022). "Evaluating the Inter-Service Communication on Microservice Architecture". In: *2022 7th International Conference on Information Technology Research (ICITR)*, pp. 1–6.

WePay (Dec. 2019). *Migrating APIs from REST to gRPC at WePay*. url: `https://wecode.wepay.com/posts/migrating-apis-from-rest-to-grpc-at-wepay`.

Wohlin, C. et al. (2012). *Experimentation in Software Engineering*. Springer Berlin Heidelberg.

Wu, Bing et al. (1997). "Legacy systems migration-a method and its tool-kit framework". In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*.

Appendix A

Systematic literature review

A.1 Search identification

A.2 Study selection

A.3 Quality assessment checklist and data extraction forms

Title and year: Migrating the Communication Protocol of Client-Server Applications, 2023

Context: A company's attempt to modernize software systems from outdated technologies to more recent approaches.

Objectives: Develop a semiautomated tool to migrate large systems used in managing different departments built with Java and Spring, from GWT-RPC and RMI to JSON over HTTP, while minimizing costs and technical debt.

Methodology: Design science research.

Data collection and analysis: The first step of the proposed migration approach involved gathering data in order to execute the migration. Such data was needed to identify service and data exchange classes. For the service classes, they were identified using coding conventions through static analysis of code. Once a service is identified, its return and parameter types can be identified as data exchange classes. To get some insights about the applications planned to undergo the migration, metrics were also collected. Those metrics were lines of code, number of service classes and methods, and data exchange classes. To evaluate the performance, the payload size, server time and user time with and without the data exchange object running were measured. All the collected data is used to assess the effectiveness of the tool created to migrate the applications and the impact of data pruning on performance.

Conclusions: A tool to migrate legacy software systems using GWT-RPC and RMI to a modern Spring HTTP API was successfully designed. Performance issues, due to data deserialization, were addressed using data exchange object pruning. However, it is admitted that RPC technologies overall perform better than REST of HTTP.

Limitations: The migrated applications still needed further validation by their owner development teams.

Title and year: Using Refactoring to Migrate REST Applications to gRPC, 2022

Context: Improve communication performance between applications.

Objectives: Develop a sufficiently general approach, applicable to any application, to refactor an existing REST application into an equivalent gRPC one. Additionally, a demonstration of the proposed approach in a small microservice application.

Methodology: Applied research with a practical example.

Data collection and analysis: The paper does not show any data collection or analysis.

Conclusions: An approach to refactor REST applications to gRPC is successfully designed. It was not intended to be used in a specific programming language or framework, but quite the opposite - is general enough to be applied to any REST application.

Limitations: The proposed refactoring approach focuses on developers to manually follow the process step by step, rather than automating some parts of it.

Database	**Search query**	**Date of search**	**Years covered**	**Number of studies**
IEEExplore	("Abstract":rest OR "Abstract":restful OR "Abstract":"rest based") AND ("Abstract":migrat* OR "Abstract":moderni* OR "Abstract":refactor*) AND ("Abstract":gRPC OR "Abstract":"remote procedure call" OR "Abstract":rpc OR "Abstract":"Apache Thrift")	22/03/2025	2018-2023	2
ACM Digital Library	[[Abstract: rest] OR [Abstract: restful] OR [Abstract: "rest based"]] AND [[Abstract: migrat*] OR [Abstract: moderni*] OR [Abstract: refactor*]] AND [[Abstract: grpc] OR [Abstract: "remote procedure call"] OR [Abstract: rpc] OR [Abstract: "Apache Thrift"]]	22/03/2025	2022	1
Science Direct	1. Title, abstract, keywords: (rest OR restful OR "rest based") AND (migration OR migrating OR migrate) AND (gRPC OR "remote procedure call" OR rpc) 2. Title, abstract, keywords: (rest OR restful OR "rest based") AND (modernize OR modernizing OR modernization) AND (gRPC OR "remote procedure call" OR rpc OR "Apache Thrift") 3. Title, abstract, keywords: (rest OR restful OR "rest based") AND (refactor OR refactoring) AND (gRPC OR "remote procedure call" OR rpc)	23/03/2025	N/A	0

Table A.1: Search execution in each academic database

Title	**Author(s) and Year**	**Source**	**Document type**	**Observations**
Migrating the Communication Protocol of Client–Server Applications	Gabriel Darbord, Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Anas Shatnawi, Abderrahmane Seriai, Mustapha Derras; 2023	ACM Digital Library	Research article	Although the study does not directly address REST to gRPC migration, the principles and challenges discussed on migrating from GWT-RPC and RMI to JSON over HTTP can offer valuable insights in such a scenario.
Using Refactoring to Migrate REST Applications to gRPC	Yunhyeok Lee, Yi Liu, 2022	ACM Digital Library	Short-paper	Directly addresses the subject under study.
A Longitude Analysis on Bitcoin Issue Repository	Chelsea Hinds-Charles, Jenelee Adames, Ye Yang, Yusong Shen, Yong Wang, 2018	IEEExplore	Conference paper	Not relevant to the subject. Excluded for not being relevant to at least one research question.

Table A.2: Study selection

	Migrating the Communication Protocol of Client-Server Applications	**Using Refactoring to Migrate REST Applications to gRPC**
Was there a clear statement of the aims of the research?	Yes	Yes
Is a qualitative methodology appropriate?	Yes	Yes
Was the research design appropriate to address the aims of the research?	Yes	Yes
Was the data collected in a way that addressed the research issue?	Yes	No
Have ethical issues been taken into consideration?	Can't tell	Can't tell
Was the data analysis sufficiently rigorous?	Yes	No
Is there a clear statement of findings?	Yes	Yes
Is the research valuable?	Yes	Yes

Table A.3: Quality assessment

Appendix B

Contract definitions

B.1 Message definitions

```
syntax = "proto3";

package petclinic.customers;

option go_package = "petclinic/grpcgateway/gen/proto/customer";
option java_package = "org.springframework.samples.petclinic.customers.
    grpc.gen.customer.types";
option java_multiple_files = true;
option java_outer_classname = "CustomerTypesProto";

message Pet {
  int32 id = 1;
  string name = 2;
  string birth_date = 3;
  PetType type = 4;
  Owner owner = 5;
}

message PetType {
  int32 id = 1;
  string name = 2;
}

message Owner {
  int32 id = 1;
  string first_name = 2;
  string last_name = 3;
  string address = 4;
  string city = 5;
  string telephone = 6;
  repeated Pet pets = 7;
}

message CreateOwnerRequest {
  string first_name = 1;
  string last_name = 2;
  string address = 3;
  string city = 4;
  string telephone = 5;
}

message UpdateOwnerRequest {
  int32 owner_id = 1;
  string first_name = 2;
  string last_name = 3;
  string address = 4;
  string city = 5;
  string telephone = 6;
}
```

B.2 Owner resource

```
syntax = "proto3";

package petclinic.customers;

option go_package = "petclinic/grpcgateway/gen/proto/customer";
option java_package = "org.springframework.samples.petclinic.customers.
    grpc.gen.customer";
option java_multiple_files = true;
option java_outer_classname = "OwnerServiceProto";

import "google/protobuf/empty.proto";
import "google/api/annotations.proto";
import "customer/Types.proto";

service OwnerService {
  rpc CreateOwner(CreateOwnerRequest) returns (CreateOwnerResponse) {
    option (google.api.http) = {
      post: "/owners"
      body: "*"
    };
  };

  rpc GetOwner(GetOwnerRequest) returns (GetOwnerResponse) {
    option (google.api.http) = {
      get: "/owners/{owner_id}"
    };
  };

  rpc ListOwners(google.protobuf.Empty) returns (GetOwnersResponse) {
    option (google.api.http) = {
      get: "/owners"
    };
  };

  rpc UpdateOwner(UpdateOwnerRequest) returns (UpdateOwnerResponse) {
    option (google.api.http) = {
      put: "/owners/{owner_id}"
      body: "*"
    };
  };
}
```

Listing B.2: Owner services

B.3 Pet resource

```
syntax = "proto3";

package petclinic.customers;

import "google/protobuf/empty.proto";
import "google/api/annotations.proto";
import "customer/Types.proto";

option go_package = "petclinic/grpcgateway/gen/proto/customer";
option java_multiple_files = true;
option java_package = "org.springframework.samples.petclinic.customers.
    grpc.gen.customer";
option java_outer_classname = "PetServiceProto";

service PetService {
  rpc GetPetTypes(google.protobuf.Empty) returns (GetPetTypesResponse) {
    option (google.api.http) = {
      get: "/petTypes"
    };
  }

  rpc CreatePet(CreatePetRequest) returns (CreatePetResponse) {
    option (google.api.http) = {
      post: "/owners/{owner_id}/pets"
      body: "*"
    };
  }

  rpc UpdatePet(UpdatePetRequest) returns (UpdatePetResponse) {
    option (google.api.http) = {
      put: "/owners/*/pets/{pet_id}"
      body: "*"
    };
  }

  rpc GetPet(GetPetRequest) returns (GetPetResponse) {
    option (google.api.http) = {
      get: "/owners/*/pets/{pet_id}"
    };
  }
}
```

Listing B.3: Pet services