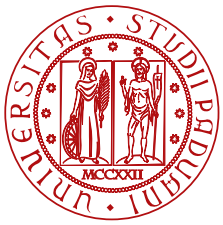

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

THE MASTER DEGREE IN COMPUTER ENGINEERING

Performance Engineering of a Microservice-Based System

Supervisor
Dr. Di Buccio Emanuele

Author
Popovic Milica

Academic Year 2024/2025

Graduation date 10/07/2025

Whether you think you can, or you think you can't – you're right.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, whose exceptional knowledge, professionalism, dedication, and constant support have meant the world to me throughout this journey. His guidance and availability were invaluable and greatly contributed to the success of this thesis.

I am also sincerely thankful to the University of Padua and its dedicated staff for providing me with the opportunity and environment to grow academically and personally.

To my beloved family—thank you for raising me with love, strength, and unwavering support. Your encouragement in every decision I make has shaped me into the person I am today.

My heartfelt thanks go to my partner, Theo, whose presence and support were a constant source of motivation. Your belief in me has been my anchor in challenging times.

Lastly, I would like to thank my friends, especially my best friend Nadja, who has been a true inspiration to me since high school. Her achievements and constant encouragement have always pushed me forward.

Without all of you, I would not be where I am today. Thank you.

Abstract

Microservices architecture is becoming increasingly popular and is widely adopted by companies due to its scalability, resilience, fault isolation, and ease of development. In today's performance-critical world, it is essential to design and implement microservices carefully, adopting best practices to ensure optimal performance. This thesis focuses on the performance engineering of a Media Monitoring application, a microservice-based system, by introducing enhancements aimed at improving efficiency, scalability, and responsiveness. Specifically, the system transitions from REST to gRPC for inter-service communication, leveraging gRPC's lower latency, higher throughput, and efficient serialization mechanisms. These changes are expected to reduce communication overhead and improve the overall performance of the system. The performance improvements are evaluated through system monitoring using Prometheus, which collects and analyzes performance metrics, and Grafana, which visualizes these metrics to provide detailed insights into resource utilization and bottlenecks. Additionally, various load tests are conducted using k6 to simulate real-life application scenarios and measure the system's behavior under different workloads. The results demonstrate the impact of these changes on the system's performance, providing insights into practices for optimizing microservice-based architectures.

Sommario

L'architettura a microservizi sta diventando sempre più popolare ed è ampiamente adottata dalle aziende per la sua scalabilità, resilienza, isolamento degli errori e facilità di sviluppo. In un mondo in cui gli aspetti legati alle performance sono sempre più centrali, è essenziale progettare e implementare i microservizi con attenzione, adottando le migliori pratiche per garantire prestazioni ottimali. Questa tesi si concentra sull'ingegnerizzazione, sfruttando informazioni sulle performance di un'applicazione di monitoraggio dei Media, in particolare un sistema basato su un'architettura a microservizi, investigando cambiamenti volti a migliorare l'efficienza, la scalabilità e la reattività. In particolare, i cambiamenti investigati riguardano il passaggio da REST a gRPC per la comunicazione tra servizi, al fine di sfruttare la minore latenza, il maggiore throughput e gli efficienti meccanismi di serializzazione di gRPC. Queste modifiche dovrebbero ridurre l'overhead di comunicazione e migliorare le prestazioni complessive del sistema. I miglioramenti delle prestazioni sono valutati attraverso il monitoraggio del sistema con Prometheus, che raccoglie e analizza le metriche delle prestazioni, e Grafana, che visualizza queste metriche per fornire informazioni dettagliate sull'utilizzo delle risorse e su eventuali colli di bottiglia. Inoltre, sono stati condotti vari test di carico con k6 per simulare scenari applicativi reali e misurare il comportamento del sistema sotto diversi carichi di lavoro. I risultati dimostrano l'impatto di queste modifiche sulle prestazioni del sistema, fornendo indicazioni sulle pratiche di ottimizzazione delle architetture basate su microservizi.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

The concept of microservices has gained significant attention in recent years from both academia and industry as a way to design and structure software applications. Microservices architecture enables building complex systems as a collection of small, independently deployable services, each focusing on a specific business capability. However, it is important to note that this architectural style is not suitable for all applications. Choosing to implement microservices requires careful consideration, particularly with respect to decisions that can impact system performance – such as the selection of development frameworks or the communication mechanisms between services. [1]

In large-scale systems composed of dozens or even hundreds of microservices, these choices become increasingly critical. This thesis focuses on analyzing the performance of microservice-to-microservice communication within a specific application. It compares two widely adopted communication protocols: REST and gRPC. A comparative analysis is conducted to identify the strengths and weaknesses of each protocol, with the aim of assisting developers in making appropriate architectural decisions.

While several studies have explored communication protocols in microservice architectures, there is still a gap in research that evaluates performance within the context of real-world applications. This thesis contributes to that subject by focusing on a real system composed of multiple microservices, aiming to improve its performance by switching from REST to gRPC for inter-service communication. The application under study is the TIPS (Technoscientific Issues in the Public Sphere) system, a microservice-based platform used for both research and teaching purposes. Although the number of services involved is relatively low, the application provides a meaningful and controlled environment for studying protocol-level performance impacts in realistic conditions. This thesis focuses on two pairs of services, which correspond to a few modules within the TIPS application. The authors of the TIPS platform plan to extend the number of modules in the future, further increasing the system's complexity and opportunities

for protocol optimization.

To evaluate the impact of this change, various test scenarios were executed using load testing as a method for analyzing how a system performs under different loads. These tests provide insight into the conditions under which gRPC is most beneficial and offer guidance on when it should be preferred over REST in practice. [6]

The main contributions of this thesis are:

- A detailed performance comparison based on real-world workloads using REST and gRPC protocols for inter-service communication, conducted on an existing microservice application used for research and education.
- Implementation and documentation of the protocol migration process within a real system.
- A set of empirical results highlighting the performance characteristics of REST and gRPC in different scenarios of internal microservice communication.

The remainder of this thesis is structured as follows:

- **Chapter 2: Background and Related Work** — Introduces key concepts related to the thesis and reviews related studies.
- **Chapter 3: Methodology** — Describes the application under study and the migration process from REST to gRPC within two use cases.
- **Chapter 4: Results and Discussion** — Explains how the test environment was configured, presents the test scenarios, provides test results, and discusses the outcomes for both use cases.
- **Chapter 5: Conclusion** — Summarizes the findings, highlights the contributions, and outlines potential directions for future research.

Chapter 2

Background and Related Work

2.1 Main Concepts

In this section the main concept essential for this thesis to be understood will be briefly explained.

2.1.1 Microservices

Microservices is a software architectural style that has gained significant popularity in recent years. It structures an application as a suite of small, independently deployable services, each running in its own process and communicating through lightweight mechanisms such as HTTP-based REST or gRPC. Each service is centered around a specific business capability and can be developed, deployed, and scaled independently. This approach contrasts with traditional monolithic architectures, where a small change often requires rebuilding and redeploying the entire application. In a microservices architecture, only the affected service needs to be updated. Moreover, services can be implemented using different programming languages, use different data storage technologies, and be managed by separate teams. This independence also enables more efficient scaling, as only the components under heavy load need to be scaled rather than the entire application. [1]

Figure 2.1 shows the fundamental architectural difference between monolithic and microservices-based systems. On the left, the monolithic application encapsulates all functionalities within a single process, and scaling is achieved by replicating the entire application. On the right, the microservices architecture splits functionalities into separate, independently deployable services. These services can be distributed and scaled individually across servers as needed, enabling greater flexibility and resource efficiency.

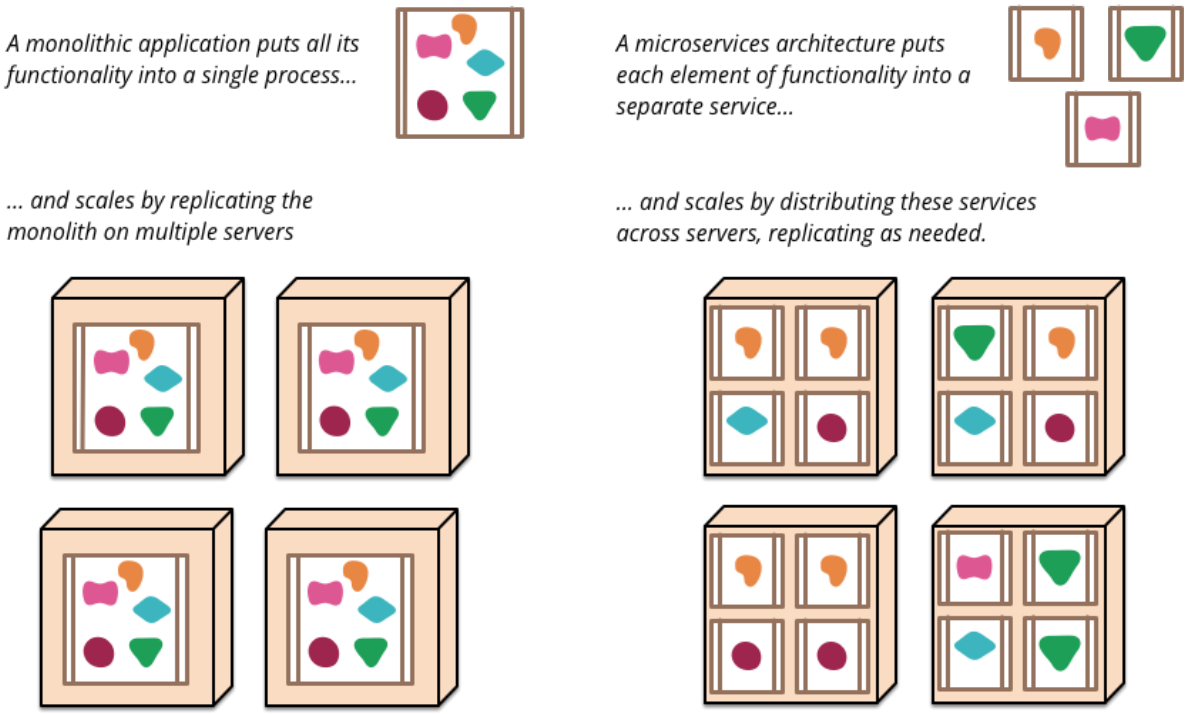

Figure 2.1: Monoliths and Microservices [1]

2.1.2 REST

Representational State Transfer (REST) is an architectural style for designing distributed systems. It was introduced by Roy Fielding in his doctoral dissertation [7] and further elaborated in subsequent works [8], [9].

Deriving REST

The design of the Web architecture is shaped by applying a series of architectural constraints to its components. By observing how each constraint influences system behavior, one can understand the properties they introduce. Adding further constraints enables the development of a refined architectural style suited for the modern Web. The derivation of REST as an architectural style follows a sequence of seven incremental refinements:

1. **Null Style:** Architectural design typically follows one of two approaches. The first begins from scratch, assembling a system piece by piece until it meets its goals. The second starts by considering the overall system requirements without constraints and progressively applies constraints to shape the architecture in a way that aligns with the system's natural behavior. REST was developed using the second, constraint-driven approach.

2. **Client-Server:** This constraint introduces a separation of concerns between clients (user

interface concerns) and servers (data storage concerns). It improves scalability, portability, and the independent evolution of client and server components.

3. **Stateless:** The communication between client and server must be stateless, meaning each request from client to server must contain all necessary information to understand the request. This constraint enhances visibility, reliability, and scalability, although it may increase per-interaction overhead when sent in a sequence of requests.

4. **Cache:** To improve efficiency, responses must be explicitly marked as cacheable or not. This constraint allows clients to reuse cacheable responses when suitable. Proper use of caching can eliminate some client-server interactions and improve performance, reducing latency and network load, but introduces risks of serving stale data.

5. **Uniform Interface:** What sets REST apart from other network-based architectural styles is its focus on a uniform interface across components. This generality simplifies the overall architecture, improves interaction visibility, and allows services and implementations to evolve independently. However, this comes at the cost of efficiency, as standardized communication may not suit all application needs. REST is optimized for transferring large-grain hypermedia data, making it ideal for the Web. To achieve this uniformity, REST relies on four key constraints: resource identification, manipulation through representations, self-descriptive messages, and hypermedia as the engine of application state.

6. **Layered System:** A REST system may be composed of hierarchical layers, where components only know about the immediate layer they interact with. This supports scalability, load balancing, and intermediary functions such as proxies and firewalls. However, it may introduce additional latency.

7. **Code-On-Demand (optional):** REST optionally allows servers to extend client functionality by sending executable code (e.g., JavaScript). This improves extensibility, but reduces visibility and thus is not a mandatory constraint [7].

While this thesis adopts the term REST to describe the HTTP-based services used, it is important to acknowledge that the endpoints in question are not fully RESTful. In [10], Richardson proposed the Richardson Maturity Model (RMM), which provides a framework for evaluating how closely an API adheres to REST principles, ranging from Level 0 (pure RPC over HTTP) to Level 3 (full HATEOAS support). This thesis refers to two HTTP-based APIs as RESTful, even though they only partially conform to REST constraints and align with different levels of the RMM. One of the future directions on the TIPS Project is to evolve the APIs toward greater RESTfulness. Further discussions on the specific RMM levels of each service are provided in Sections 3.3 and 3.4, where the HTTP-based services are examined in more detail.

Resources, Resource Identifiers and Representations

In REST, the fundamental abstraction of information is a `resource`, which refers to any piece of information that can be named—ranging from documents and images to services or even physical entities like people. Resources are identified by unique resource identifiers (URIs). The client interacts with the resource using its `representation`, which is typically formatted as JSON or XML. Representations carry both the data and metadata necessary to describe the resource [7].

REST and HTTP are not the same

While REST commonly uses HTTP as the underlying protocol, REST and HTTP are not synonymous. REST is an architectural style that can be implemented over other protocols besides HTTP, such as WebSockets or FTP. Similarly, HTTP can be used in non-RESTful ways, such as in SOAP-based web services. REST uses HTTP because it aligns well with REST principles, particularly with the standardized methods and stateless communication [11].

Figure 2.2 shows the basic interaction model in a RESTful architecture. REST clients, such as desktop and mobile devices, send REST requests to a REST server. These requests consist of an HTTP method (e.g., GET, POST) and a URI that identifies the resource. The REST server processes the request and returns a REST response containing the requested resource representation in either XML or JSON format.

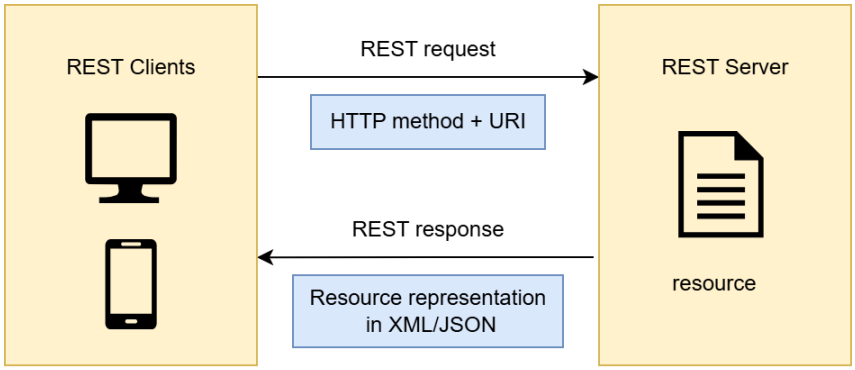

Figure 2.2: REST in action

2.1.3 gRPC

gRPC is a high-performance, open-source Remote Procedure Call (RPC) framework initially developed by Google. It allows applications to communicate with each other across different environments and languages through a well-defined service interface. gRPC uses HTTP/2 as its

transport protocol and Protocol Buffers (protobufs) as the interface description language (IDL), which together provide a robust platform for building distributed systems.

Service Definition

Services in gRPC are defined in `.proto` files using Protocol Buffers. These definitions specify service methods, their parameters, and return types. Once compiled, these files generate code in multiple languages (e.g., Java, Go, Python, C++). The following is an example of a service definition with its parameters and return types.

```
service TipsService {
  rpc ExtractNER(ArticleContent) returns (NERResult);
}

message ArticleContent {
  string title = 1;
  string body = 2;
}

message NERResult {
  map<string, string> title = 1;
  map<string, string> body = 2;
  string language = 3;
}
```

Once a `.proto` file is created, the protocol buffer compiler for various programming languages will generate client- and server-side code. Typically, the server implements the methods declared in the `.proto` file and runs a gRPC server to accept client calls. On the other hand, the client creates a local object called a stub, on which the defined methods are called.

Service Method Types

gRPC supports four types of service methods, which allow flexibility in how clients and servers communicate:

- **Unary RPC**: Client sends a single request and gets a single response.
- **Server Streaming RPC**: Client sends one request and receives a stream of responses.
- **Client Streaming RPC**: Client sends a stream of requests and gets a single response.

- **Bidirectional Streaming RPC**: Both client and server send a stream of messages simultaneously [12].

Figure 2.3 shows the service method types.

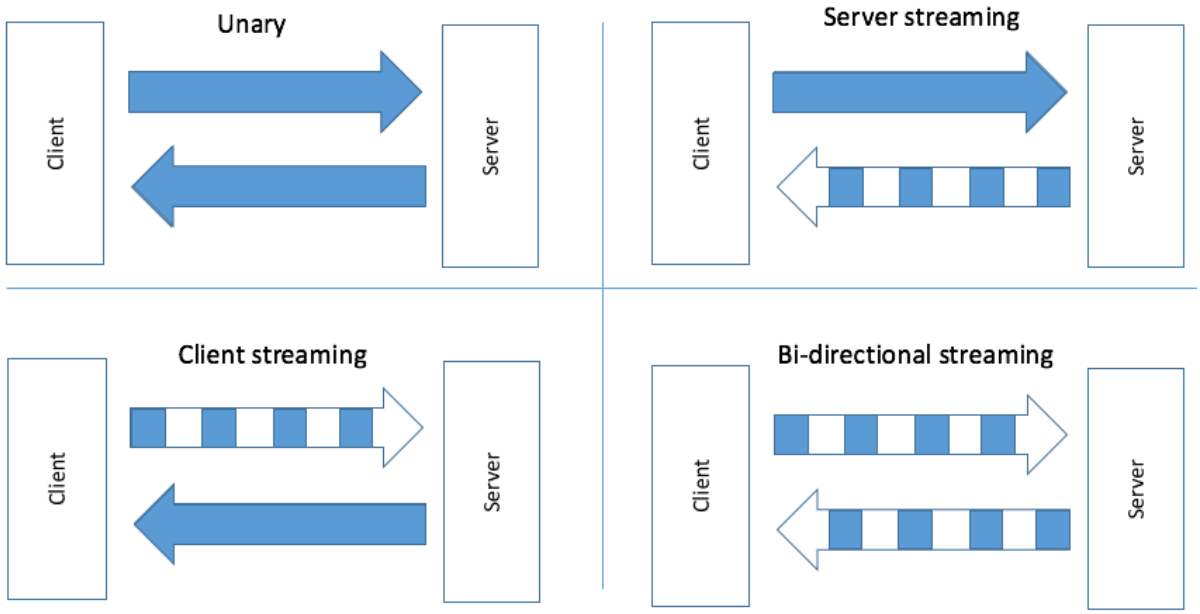

Figure 2.3: gRPC Service Method Types [2]

This project uses only Unary RPCs, the simplest gRPC communication model. This choice was made to ensure consistency, simplicity, and fairness in comparing the performance of gRPC and REST. In particular, streaming RPCs in Python tend to be slower due to the overhead of creating additional threads for handling incoming and/or outgoing message streams, making them less efficient compared to other supported languages. Additionally, streaming RPCs introduce added complexity and are better suited for use cases requiring long-lived, continuous data flows, which were not needed in this project [13].

2.1.4 Comparing REST and gRPC

In the previous two subsections, two popular architectural styles for communication in distributed systems were discussed: gRPC and REST. This subsection highlights their key differences and the scenarios in which each is most suitable.

Both gRPC and REST are designed to enable communication between distributed software components. They share several similarities:

- Both follow a stateless client-server communication model.
- Both commonly use HTTP as the underlying transport protocol.

- Both support cross-language implementations.

However, they differ significantly in design philosophy, data serialization, and supported communication patterns.

In gRPC, the client invokes a specific function on the server as if it were a local method call. The request includes the method name and any required parameters. gRPC uses HTTP/2 as its transport protocol, enabling advanced features such as multiplexed streams and bidirectional communication.

REST, in contrast, is typically built on HTTP/1.1 and revolves around the concept of manipulating resources via standard HTTP methods (GET, POST, PUT, DELETE). It is based on an entity-oriented design, where clients interact with resource URIs rather than calling named functions.

Example – REST vs. gRPC:

- gRPC (service-oriented): `getDocuments(documentsRequest) → result`
- REST (entity-oriented): `POST /documents (documentsRequest) → result`

In terms of data representation, REST typically uses human-readable JSON for data exchange, which is flexible but requires more processing time for serialization and deserialization. gRPC, on the other hand, uses Protocol Buffers by default, which offer a compact, binary format that is faster and more efficient but less human-readable.

Beyond the differences in communication and serialization, REST and gRPC diverge in several additional technical aspects.

The coupling between client and server is also handled differently. REST is loosely coupled: clients can work with the server as long as they understand the resource structure and HTTP semantics, making the system more adaptable to changes over time. gRPC, on the other hand, introduces tighter coupling since both sides depend on the shared `.proto` specification. Any changes to the service definition require regenerating and updating client and server code, requiring greater coordination.

Code generation is another area where gRPC stands out. With the help of the `protoc` compiler, gRPC can automatically generate both client-side and server-side code in various programming languages, making the development process faster. REST lacks a built-in mechanism for code generation; developers typically rely on third-party tools or write the integration logic manually.

Finally, when it comes to streaming capabilities, REST is limited to one-time request-response cycles. gRPC, with its support for bidirectional streaming over HTTP/2, allows both

client and server to exchange multiple messages in real time. This feature is especially valuable for applications requiring continuous data flow [14].

Suitability and Use Cases

The selection between REST and gRPC largely depends on the application context, required performance, and system design.

REST is a better fit when the system is web-based and interacts directly with browsers or public clients. It is suitable when API consumers value readability and intuitive interfaces, when data exchanges are relatively lightweight and infrequent, and when simplicity, ease of debugging, and compatibility with standard web tools are key concerns.

gRPC is preferred when the system comprises performance-critical internal services or backend components. It is the better choice when there is a need for efficient data transmission or real-time bidirectional communication, when the application spans multiple services and languages—especially in a microservice architecture—and when the interface contracts are well-defined and relatively stable over time [14].

2.1.5 Performance Testing

Testing plays a crucial role in software development, as it ensures the quality, reliability, security, and overall performance of a system. While many different types of testing exist, this subsection focuses on performance testing and its subset — load testing. These types were selected because they provide direct insights into how a system or component behaves under varying levels of demand, which is essential for comparing the performance of the two protocols evaluated in this thesis. Performance testing evaluates the efficiency of a system to ensure it remains responsive and stable under different load conditions in a production-like environment. Key metrics often monitored during performance testing include throughput, latency, response times, and CPU usage. These metrics help developers identify performance bottlenecks and areas for optimization, providing a deeper understanding of how the system reacts to specific workloads.

To ensure the validity of performance tests, it is important to simulate realistic production conditions. This means that the actual workload must be known prior to testing. Furthermore, performance tests should involve a gradual increase in load, allowing developers to observe how the system scales and to identify bottlenecks or failure points during each phase of testing.

Monitoring the system's stability, performance, and resource consumption is essential throughout the testing process. Results should be documented and maintained for future reference and iterative improvements [15].

Load Testing

Load testing is a subset of performance testing, specifically aimed at evaluating a system's behavior under expected peak user activity. The main goal of load testing is to assess the system's scalability and capacity, ensuring it can handle real-world usage levels. This type of testing helps track how system resources and internal processes behave under stress from concurrent users or operations. Load tests are typically conducted using incremental load patterns, similar to performance testing. The load is gradually increased to simulate realistic user growth and to observe how the system scales. By performing load testing early, developers can determine whether the system performs adequately during peak usage and identify potential bottlenecks or inefficient components. These insights allow teams to optimize performance before deployment to a production environment [15]. To conclude, Performance testing is a broader term that tests the overall speed, responsiveness, stability, and resource usage of a system under various conditions while load testing is a specific type of performance testing that simulates real-world peak usage to determine how the system handles expected traffic levels.

k6 Load Testing Tool

Grafana k6 is an open-source, developer-friendly load testing tool designed to test and improve the performance and reliability of applications [5]. It supports various types of testing such as load testing, integration testing, infrastructure testing, and more. k6 allows developers to simulate realistic user scenarios and observe how the system behaves under varying loads. This helps identify bottlenecks and performance issues before deploying the system to production. The configuration of k6 tests is highly customizable. Developers can define parameters such as the number of virtual users (VUs), the number of iterations, request durations. Test scripts in k6 are written in JavaScript or TypeScript, which makes the syntax familiar and easy to adopt for most developers. A k6 script typically consists of two main components:

- **Imports** – Used to include additional k6 modules or JavaScript libraries, such as the HTTP module.
- **Default function** – The core test logic is written here. It is executed for each virtual user.

Listing 1 presents a basic k6 test script that performs 10 HTTP GET requests to a URL and waits one second between each request.

```
1 import http from 'k6/http';
2 import { sleep } from 'k6';
3
4 export const options = {
```

```
  iterations: 10,
};

export default function () {
  http.get('https://quickpizza.grafana.com');
  sleep(1);
}
```

Listing 1: Basic k6 load test example [5]

To run a test locally, the command in Listing 2 is used. If you have a k6 Cloud account, tests can be outsourced to the cloud by using the command in Listing 3.

```
k6 run script.js
```

Listing 2: Run k6 test locally

```
k6 cloud run script.js
```

Listing 3: Run k6 test on k6 Cloud

During test execution, k6 runs multiple iterations in parallel using Virtual Users (VUs). At the end of the test, a summary report is printed to `stdout`. This includes metrics like `http_req_duration`, which shows the total latency of HTTP requests. Developers can also define custom metrics, thresholds, and checks [5].

This section only scratches the surface, but k6 supports many more advanced features [5].

2.1.6 Monitoring and Observability

Continuous monitoring of systems is crucial for building robust microservice architectures, as it helps prevent and handle errors, and improves overall performance. Monitoring enables developers to detect issues as soon as they occur and respond accordingly. There are many tools available for application monitoring, but a widely adopted combination is Prometheus[1] and Grafana[2] [16]. While monitoring focuses on collecting and storing data, observability goes a step further by helping developers interpret and understand that data. It makes the internal state and behavior of the system transparent and easier to diagnose [3].

[1] `https://prometheus.io/`
[2] `https://grafana.com/`

Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit designed for reliability and scalability. It collects and stores metrics as time series data, which consist of a timestamp and optional key-value pairs called labels. Prometheus gathers these metrics by scraping predefined HTTP endpoints exposed by monitored targets. These metrics are essential for understanding application performance and may include request durations, latency, error rates, and resource usage. The core component of the Prometheus ecosystem is the Prometheus server, which is responsible for scraping and storing time series data. Another important component is the set of client libraries used to instrument application code. These libraries allow developers to define and expose custom metrics, enabling Prometheus to collect detailed insights from running services.

Prometheus also offers a built-in expression browser, available at the ‘/graph‘ endpoint on the Prometheus server. This browser allows users to enter PromQL (Prometheus Query Language) expressions and view the results in table or graph form. For example, Listing 4 shows how to return all time series with the metric `http_requests_total`, while Listing 5 filters by the job label `apiserver`. PromQL supports powerful features such as regular expressions, subqueries, functions, and more.

Another notable feature of Prometheus is its alerting system. Developers can define alerting rules in the Prometheus server, which then sends alerts to the `Alertmanager`. The Alertmanager is responsible for managing alert notifications and can integrate with email, messaging platforms, and other channels [17].

```
1 http_requests_total
```

Listing 4: Prometheus example 1

```
1 http_requests_total{job="apiserver"}
```

Listing 5: Prometheus example 2

Grafana

Grafana is an observability platform that integrates well with Prometheus. Observability is the process of making a system’s internal state transparent by exposing and visualizing the data it produces, enabling users to determine whether the system is healthy and functioning as expected. This is particularly important in microservice-based architectures, where understanding the relationships and health of individual services is critical. Once Prometheus collects and stores the

metrics, Grafana allows developers to create rich visualizations for various use cases. Available visualization types include time series graphs, bar charts for categorical data, histograms to show value distributions, tables, logs, and more. Users can combine multiple panels into custom dashboards, tailored for specific monitoring goals. Grafana also supports alerting, allowing alerts to be displayed on dashboards or sent through external notification channels [3].
Figure 2.4 illustrates a sample Grafana dashboard.

Figure 2.4: Example Grafana dashboard [3]

2.2 Related Work

There have been multiple research papers focusing on microservices performance, testing strategies, and communication protocols. For instance, [18] presents a set of challenges associated with testing in microservice-based systems, such as instability in execution environments and variability in test results. To address these issues, the authors propose characteristics of an ideal

testing environment, which include the use of realistic workloads, test executions at both unit and integration levels, and readily available performance metrics. This thesis adopts these principles as a foundation for evaluating performance in a controlled and consistent manner.

The study in [19] explores the influence of microservices architecture on system performance and proposes several strategies for enhancement, including efficient inter-service communication mechanisms like gRPC, service scaling, load balancing, caching, and performance monitoring. Additionally, containerization is identified as a valuable practice for isolating and optimizing microservices. This thesis builds upon those findings by incorporating gRPC communication, Docker-based containerization, and integrated performance monitoring to assess the impact of protocol migration.

Testing approaches in microservice environments are further discussed in [20], which analyzes testing difficulties and potential solutions. The authors investigate the relationship between specific testing challenges, corresponding solutions, and quality attributes. While this thesis is not centered around testing methodology, it benefits from these insights to design a structured testing strategy.

In terms of development frameworks, [21] evaluates multiple frameworks for building microservices-based applications, focusing on aspects such as design patterns, features, and performance. The study shares similarities with this thesis in that it explores different technologies for improving inter-service communication.

The comparative analysis conducted in [22] investigates synchronous REST and asynchronous RabbitMQ for communication between services. The authors use JMeter for performance testing and apply the evaluation to an application. Although the study shares a similar objective with this thesis—evaluating the efficiency of communication protocols—it differs significantly in complexity. The application in [22] is minimalistic, involving only a pair of services with simple string transmission and limited logic. In contrast, this thesis evaluates production-like microservices with more complex data structures and processing logic, offering a more realistic performance evaluation scenario.

Several works have focused specifically on the migration from REST to gRPC. For example, [23] offers a practical overview of the migration process, which was valuable in guiding the early phases of this work. However, the focus in that study is solely on migration steps and not on empirical performance evaluation. A related industry case study by WePay [24] explains why gRPC was adopted over REST and describes their internal migration strategy. While informative, their study lacks experimental testing or performance measurements.

The most technically aligned work is [25], which compares REST and gRPC within a microservice-based application. The author evaluates the protocols in terms of latency and throughput but intentionally excludes business logic, database operations, and logging to isolate

communication performance. The results indicate that gRPC is 7 to 10 times more performant than REST, depending on the scenario, although REST was four times faster to implement. In contrast, this thesis includes realistic application components and operational context, offering a more practical view of performance differences in a near-production environment.

To the best of our knowledge, no prior work included both the process of migrating from REST to gRPC for inter-service communication with a performance comparison conducted within a real-world application setup. The goal of this thesis is to fill that gap by implementing such a migration and evaluating the results, thereby contributing both methodological insights and practical performance benchmarks.

Chapter 3

Methodology

This chapter outlines the methodology adopted to evaluate the performance of REST and gRPC communication protocols in a microservice-based environment. It introduces the TIPS system, a real-world application used in research and teaching, which serves as the experiment for this study. The chapter describes the system architecture, the approach used to migrate inter-service communication from REST to gRPC, and the configuration of performance testing tools.
The methodology is organized around two key use cases within TIPS: the `collector-processor` communication and the `api-tm` communication.
The chapter is structured as follows:

- Section 3.1 reports a concise description of the TIPS Project and of the platform developed to achieve this objective;
- Section 3.2 reports an overview of the system's microservice-based architecture and deployment using Docker;
- Section 3.3 describes the first communication pair, which consists in the Collector and Processor services (`collector-processor` communication);
- Section 3.4 describe services involved in the Topic Modeling use case and their communication (`api-tm` communication).

3.1 TIPS Project

This section briefly introduces the architecture of the TIPS project, which serves as the basis for testing and evaluation in this thesis. The TIPS project focuses on creating and testing automated methods for collecting, classifying, and analyzing digital content found online — particularly from news websites and social media platforms. Its goal is to track developments and trends

in science and technology-related topics. To achieve this, TIPS brings together theories, expertise, and research approaches from several areas within the social sciences, including Public Communication of Science and Technology (PCST), Science and Technology Studies (STS), digital methods, content analysis, social representations, and information and communication technologies (ICT). With its interdisciplinary approach, the project includes researchers from various academic backgrounds. TIPS is part of the broader research initiatives conducted by the Pa.S.T.I.S. Group.

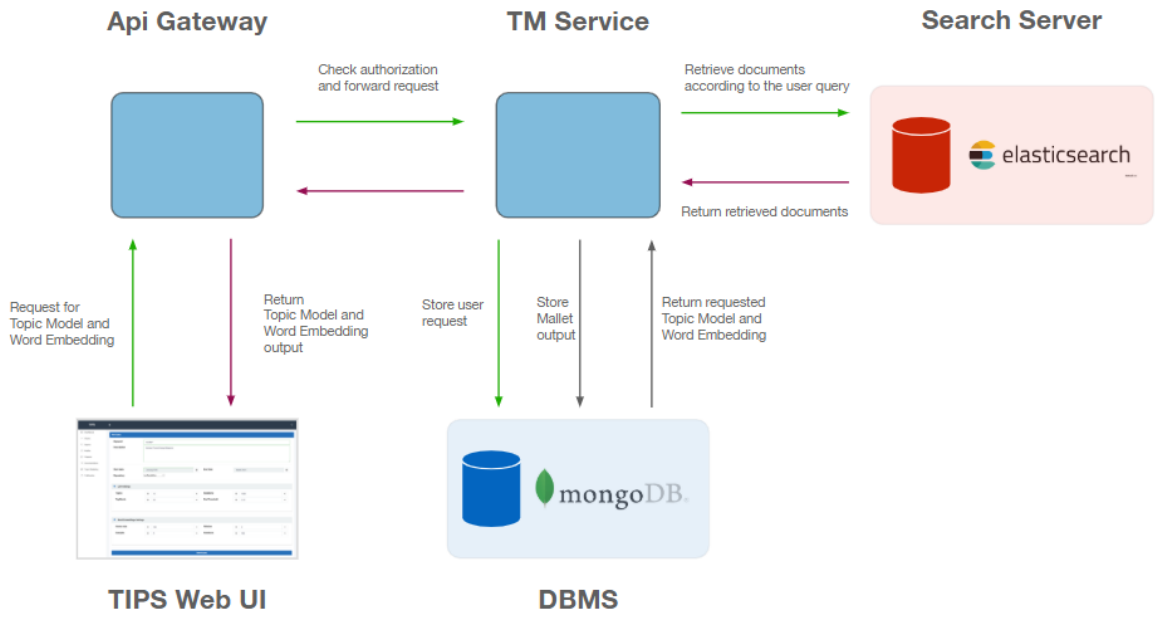

Figure 3.1: TIPS Architecture [4]

Figure 3.1 shows the main modules that make up the TIPS system. The system provides a user-facing interface (TIPS Web UI) through which users can request topic modeling results based on specific query parameters.

At the heart of the architecture lies the `TM` service, which handles the main business logic related to topic modeling and document retrieval. These operations are triggered via requests routed through the `Api Gateway`, which is responsible for handling authentication and forwarding requests to backend services. The API layer ensures secure and organized access to the underlying system components.

Document retrieval is carried out by querying the `Search Server`, implemented using Elasticsearch. The TM Service queries Elasticsearch to retrieve documents relevant to a user's query and then returns the results through the API Gateway to the frontend.

All user queries and modeling outputs are stored in a NoSQL database (MongoDB), which acts

as the system's primary persistent store.
Although the figure focuses on the modeling and interaction phase of the TIPS system, it is important to note that earlier versions—and the backend infrastructure—still include a sophisticated pipeline for collecting and enriching documents. This includes:

- **Harvesting**: Collecting documents from various sources (e.g., RSS feeds from blogs or newspapers), while capturing metadata such as publication date and author.
- **Scraping**: Extracting the main content of articles and removing extraneous web elements like ads and banners.
- **De-duplication and Near-Duplicate Detection**: Identifying and eliminating exact and near-duplicate articles to ensure data quality and avoid redundancy.
- **Named Entity Recognition (NER)**: Detecting relevant entities (e.g., names of people, organizations, places).
- **Classification and Indicator Assignment**: Classifying articles by thematic categories and assigning indicators that quantify the prominence of certain technoscientific or societal characteristics (e.g., risk, individualization).

These enrichment and classification steps are mostly managed by background services and libraries such as `hactar` and `tips-data`, and the resulting indexable documents are made searchable via Elasticsearch.

3.2 TIPS Architecture and Deployment

The modules shown in Figure 3.1 are implemented using a microservices architecture. These services are deployed and managed using Docker Compose, which defines the services corresponding to the modules described earlier. Docker Compose configuration file enables defining and running multi-container applications, thereby simplifying and accelerating the development process. [26]
Listing 6 presents the Docker Compose configuration used to build and run the TIPS system. The configuration includes the following services:

- **MongoDB** – Stores application data, including documents.
- **Elasticsearch** – Provides search and indexing capabilities.
- **API Gateway** – Acts as the access point for web interfaces and APIs.

- **Collector** – Implements document harvesting, de-duplication, storage, and indexing.
- **Topic Modeling (TM)** – A Java-based module that performs topic modeling and communicates with the MongoDB and Elasticsearch services.

All services are interconnected via a virtual Docker network named `tips-net`. Dependency checks are configured to ensure that MongoDB and Elasticsearch are ready before dependent services start.

```yaml
name: tips-services
services:
  mongo:
    image: mongo:4.0.28
    container_name: tips-mongodb
    ports:
      - 27017:27017
    networks:
      - tips-net

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.5.4
    container_name: tips-elastic
    ports:
      - 9200:9200
    networks:
      - tips-net

  api:
    build: ../tips-api-gateway
    container_name: tips-api-gateway
    networks:
      - tips-net
    depends_on:
      mongo:
        condition: service_healthy
      elasticsearch:
        condition: service_started

  collector:
    build: ../tips-collector-service/collector
```

```
    container_name: tips-collector
    networks:
      - tips-net
    depends_on:
      mongo:
        condition: service_healthy
      elasticsearch:
        condition: service_started

  tm:
    build: ../tips-tm-service
    container_name: tips-tm
    networks:
      - tips-net
    depends_on:
      mongo:
        condition: service_healthy
      elasticsearch:
        condition: service_started
    ports:
      - 8080:8080

networks:
  tips-net:
    driver: bridge

volumes:
  mongodb-data:
```

Listing 6: Docker Compose file defining TIPS microservices

3.3 Collector - Processor Use Case

The Collector module includes a Named Entity Recognition (NER) process that identifies people, locations, and organizations [4]. This process is computationally intensive and can be implemented using libraries such as spaCy[1] or Stanza[2]. For this thesis, spaCy was chosen for

[1] https://spacy.io
[2] https://stanfordnlp.github.io/stanza/

its speed, which shortens test execution time. TIPS already integrates spaCy directly into the collector service.

However, it may be beneficial to extract this functionality into a separate processor service. Communication between collector and processor could use REST, as with other internal TIPS communications, but this study evaluates whether using gRPC can provide performance improvements.

To conduct the performance comparison, an additional processor service is added to the Docker Compose setup. This processor service implements both gRPC and REST servers, allowing direct comparison between the two communication protocols.

The configuration in Listing 7 adds these servers. Both are part of the same image, but run on separate ports: gRPC on port 50051 and REST on port 5000.

```
grpc-server:
    build:
      context: ../tips-processor-service
    container_name: tips-grpc-server
    networks:
      - tips-net
    ports:
      - 50051:50051

rest-server:
    build:
      context: ../tips-processor-service
    container_name: tips-rest-server
    networks:
      - tips-net
    ports:
      - 5000:5000

```

Listing 7: Added gRPC and REST servers to Docker Compose

The method used for comparison is named *extractNER*. The collector service calls this method by passing the title and body of an article. The processor service then performs the extraction logic and returns a result composed of the processed title, body, and the language used. The sequence diagram in Figure 3.2 illustrates the communication between the services.

A typical use case of this method will be briefly explained to give the reader a better understanding of its performance-critical nature. Within the collector service, there is a scheduler that

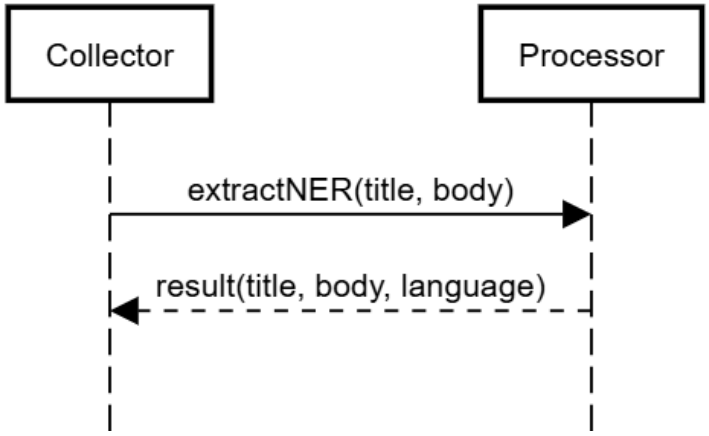

Figure 3.2: Communication between the collector and the processor services

runs a daily job responsible for gathering around 1500 documents — the exact number may vary depending on the supported languages, as TIPS handles multiple languages. For each of these documents, NER extraction is performed so performance is crucial.

In addition, there are scenarios where the entire document corpus must be reprocessed and re-indexed. In such cases, millions of documents are sent to the processor service to perform NER extraction. These examples highlight the need for the extraction to be as fast and efficient as possible.

The candidates for NER extraction are the titles and bodies of the collected documents, and each document results in one request. Listing 8 shows an example of such a request, while Listing 9 presents a corresponding example of the response.

```
{
  "title": "Hungary’s crackdown on LGBTQ+ content violates human rights, says
  ↪ EU lawyer",
  "text": "A Hungarian law banning content about LGBTQ+ people from schools
  ↪ and primetime TV has been found to violate basic human rights and
  ↪ freedom of expression by a senior legal scholar at the European court of
  ↪ justice. The non-binding opinion from the court’s advocate general,
  ↪ Tamara Ćapeta, issued on Thursday, represents a comprehensive demolition
  ↪ of the arguments made by the Hungarian government defending its
  ↪ so-called childprotection law, passed in 2021."
}
```

Listing 8: An example of a NER request

```
{
  "title": {
```

```
    "Hungary": "GPE",
    "EU": "ORG"
  },
  "body": {
    "2021": "DATE",
    "Tamara Ćapeta": "PERSON",
    "Thursday": "DATE",
    "Hungarian": "NORP",
    "European": "NORP"
  },
  "language": "en_core_web_sm"
}
```

Listing 9: An example of a NER response

3.3.1 Monitoring and Observability Services

To enable performance comparison between gRPC and REST within this use case, the monitoring tools Prometheus and Grafana – explained in Section 2.1.6 – are added to the Docker Compose configuration. Listing 10 shows the additional services integrated to support this comparison. Prometheus is responsible for collecting and storing metrics from the services under test, while Grafana provides visualization capabilities, allowing those metrics to be displayed through customizable dashboard panels.

```
grafana-server:
    build: ./docker/grafana
    container_name: grafana-server-tips
    networks:
      - tips-net
    ports:
      - 3030:3000

prometheus-server:
    build: ./docker/prometheus
    container_name: prometheus-server-tips
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
    networks:
      - tips-net
```

```
    ports:
      - 9091:9090
```

Listing 10: Added services for monitoring and observability

3.3.2 Conversion Steps

Firstly, the NER logic was moved to the newly created REST server and tested to confirm everything works same as before. The second step includes creation of gRPC server and its NER method. The following steps were conducted to refactor the REST method to gRPC:

- **REST API Endpoint Identification**
  The first step was to identify the REST API endpoint to be converted. In our case, the endpoint is `/extract-ner`. Listing 11 shows the implementation of this endpoint, while Listing 12 shows how the collector service calls it and parses the result.

```
nlp = spacy.load("en_core_web_sm")

REQUEST_COUNT = Counter('extract_ner_requests_total', 'Total ExtractNER
↪  requests')
REQUEST_LATENCY = Histogram('extract_ner_latency_seconds', 'Latency for
↪  ExtractNER requests')
FAIL_COUNT = Counter('extract_ner_failed_total', 'Total failed
↪  ExtractNER requests')

start_http_server(8000)

app = Flask(__name__)

@app.route('/extract-ner', methods=['POST'])
def extract_ner():
    REQUEST_COUNT.inc()
    with REQUEST_LATENCY.time():
        try:
            req_data = request.get_json()

            title = req_data.get("title", "")
            body = req_data.get("body", "")
            language = "en_core_web_sm"
```

```
21
22              title_ents = {ent.text: ent.label_ for ent in
                ↪  nlp(title).ents} if title else {}
23              body_ents = {ent.text: ent.label_ for ent in nlp(body).ents}
                ↪  if body else {}
24
25              return jsonify({
26                  "title": title_ents,
27                  "body": body_ents,
28                  "language": language
29              })
30          except Exception as e:
31              FAIL_COUNT.inc()
```

Listing 11: REST implementation of extract_ner method

```
1 url = "http://tips-rest-server:5000/extract-ner"
2 response = requests.post(url, json={"title": title, "body": content})
3 ner = response.json()
```

Listing 12: The collector's call to extract-ner REST endpoint

Note that the REST endpoint exposed by the processor service, which handles NER extraction, is not fully RESTful in the strictest sense. It uses an action-based URI (/extract-ner) and accepts HTTP POST requests with JSON content containing the text to be analyzed. According to the Richardson Maturity Model (RMM) [10], this design corresponds to Level 0, as it leverages HTTP only as a transport mechanism without exposing identifiable resources or leveraging the semantics of HTTP methods beyond POST. Although functionally effective, this approach does not implement resource-oriented principles or hypermedia controls. Thus, while the endpoint is HTTP-based and is often loosely referred to as REST, it does not fully comply with REST constraints. A more RESTful design would involve using resource-based URIs and structured responses with links. This thesis acknowledges this limitation and uses the term REST in a broader sense to describe HTTP-based APIs.

- **gRPC Installation**

  For Python, the required dependencies are `grpcio` and `grpc-tools`. The former provides the core gRPC library, while the latter includes the Protocol Buffers compiler `protoc` and a plugin for generating client and server code from `.proto` files [27]. Note that the installation process may vary depending on the programming language used.

- **Protocol Buffers Definition**

  As mentioned in Section 2.1.3, services in gRPC are defined using Protocol Buffers in a `.proto` file. Listing 13 shows the definition of the gRPC method corresponding to the REST endpoint, including the request and response message types.

```
syntax = "proto3";

service TipsService {
  rpc ExtractNER (ArticleContent) returns (NERResult);
}

message ArticleContent {
  string title = 1;
  string body = 2;
}

message NERResult {
  map<string, string> title = 1;
  map<string, string> body = 2;
  string language = 3;
}
```

Listing 13: gRPC ExtractNER method definition

- **gRPC Code Generation**

  The next step is to run a command that generates code based on the `.proto` file. This command generates two files: `tips_pb2.py`, which contains the request and response classes, and `tips_pb2_grpc.py`, which contains the client and server classes [27]. The command used is shown in Listing 14.

```
RUN python -m grpc_tools.protoc -I./proto --python_out=. --pyi_out=.
↪  --grpc_python_out=. ./proto/tips.proto
```

Listing 14: gRPC code generation for Python

- **gRPC Method implementation**

  After the gRPC code from the previous step is generated, the next task is implementing the *ExtractNER* method, which is equivalent to the REST method shown in Listing 11. The gRPC implementation is shown in Listing 15.

```
nlp = spacy.load("en_core_web_sm")

```

```
REQUEST_COUNT = Counter('extract_ner_requests_total', 'Total ExtractNER requests')
REQUEST_LATENCY = Histogram('extract_ner_latency_seconds', 'Latency for ExtractNER requests')
FAIL_COUNT = Counter('extract_ner_failed_total', 'Total failed ExtractNER requests')

start_http_server(8050)

class TipsServiceServicer(tips_pb2_grpc.TipsServiceServicer):
    def ExtractNER(self, request, context):
        REQUEST_COUNT.inc()
        with REQUEST_LATENCY.time():
            try:
                language = "en_core_web_sm"

                title_ents = {ent.text: ent.label_ for ent in nlp(request.title).ents} if request.title else {}
                body_ents = {ent.text: ent.label_ for ent in nlp(request.body).ents} if request.body else {}

                return tips_pb2.NERResult(
                    title=title_ents,
                    body=body_ents,
                    language=language
                )
            except Exception as e:
                FAIL_COUNT.inc()
```

Listing 15: gRPC ExtractNER method implementation

- **gRPC Server creation**

  To make the gRPC method functional, a gRPC server must be created. Listing 16 shows the code required to start the server. More precisely, the code creates a gRPC server, registers the *TipsServiceServicer*, binds it to port 50051, and starts it.

```
def serve_grpc():
    num_workers = multiprocessing.cpu_count()
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=num_workers))
```

```
    tips_pb2_grpc.add_TipsServiceServicer_to_server(TipsServiceServicer
    ↪ (),
    ↪ server)
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()
```

Listing 16: gRPC server setup for Python

- **gRPC Client creation**
  The final step of the conversion process is to create a gRPC client. To communicate with the gRPC server, a client (also called a stub) must be created. Using the stub, the client can send requests to the server. The client-side workflow includes:
  - Opening a gRPC channel with the URL and port used by the gRPC server,
  - Creating a stub to interact with the server and
  - Calling the gRPC method with the appropriate request message [28].

  Listing 17 shows the channel and stub creation, while Listing 18 demonstrates a method call that mirrors the REST version in Listing 12. Code withing Listing 18 sends a request using the stub and converts the response to a dictionary format. Note that the channel and stub are created only once, while the *ExtractNER* method is invoked once for each document. After the conversion, both REST and gRPC responses were compared to ensure consistency.

```
channel = grpc.insecure_channel("tips-grpc-server:50051")
stub = tips_pb2_grpc.TipsServiceStub(channel)
```

Listing 17: gRPC client setup for Python

```
ner_proto = stub.ExtractNER(tips_pb2.ArticleContent(title=title,
↪ body=content))
ner = MessageToDict(ner_proto)
```

Listing 18: The collector's call to the gRPC method

3.4 Api - TM Use Case

Section 3.3 described one scenario of transitioning from REST to gRPC. This section presents a second use case, focusing on the communication between the API gateway (api) and the topic

modeling (tm) modules within TIPS. These services are referred to as `api` and `tm` in the Docker Compose file in Listing 6, and will be called as such from now on. Presenting two use cases demonstrates how gRPC facilitates communication between services implemented in different programming languages. While the `collector` and `processor` services are implemented in Python, the `api` service is written in Node.js and the `tm` service in Spring (Java). In this scenario, `api` acts as the client and `tm` as the server.

The sequence diagram in Figure 3.3 illustrates the communication between these services. The conversion method, called `getDocuments`, retrieves a list of documents based on query and topic parameters. It first fetches documents related to the topic from MongoDB, then uses those document IDs to retrieve the full documents from Elasticsearch within the specified index range. The conversion steps are detailed in the following subsection. Listing 19 shows an example of a `getDocuments` request, while Listing 20 shows a corresponding example of the response. The request asks for only two documents, but the array length of the top documents could be much longer. The response contains elements with a `text` field, which is truncated here for readability. In contrast to the Collector - Processor use case, where the `extractNER` method is called around 1500 times in one batch, the `getDocuments` method is invoked as needed, depending on the specific scenario, and is not called as frequently.

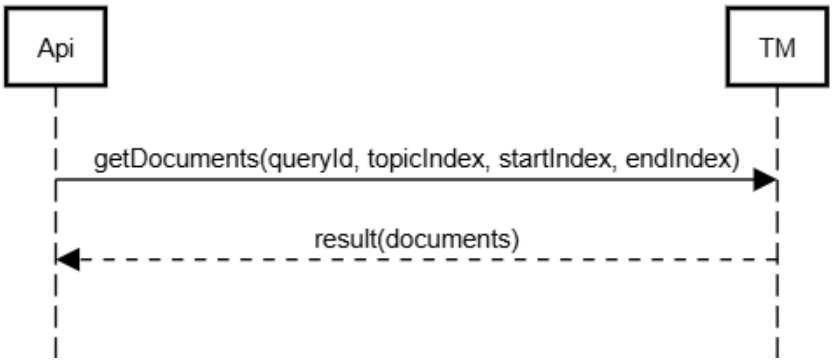

Figure 3.3: Communication between the api and the tm services

```
{
    "queryId": "6835aab0c8dd352b32858a56",
    "topicIndex": 1,
    "startIndex": 0,
    "endIndex": 2
}
```

Listing 19: An example of getDocuments request

```
[
    {
        "id": "682596f8cb540dcb19b77b7a",
        "text": "Poet and artist Jay Bernard: ‘I think our problem today is
        ↪  technical illiteracy’ ~Born in 1988, Jay Bernard is a British
        ↪  poet, artist and film programmer born in London. Their multimedia
        ↪  project Surge: Side A, about the 1981 New Cross...",
        "newspaper": "GUARDIAN",
        "section": "api|GUARDIAN|Art and design",
        "date": "2025-02-09",
        "url": "https://www.theguardian.com/artanddesign/2025/feb/09/poet-an
        ↪  d-artist-jay-bernard-i-think-our-problem-today-is-technical-illi
        ↪  teracy",
        "title": "Poet and artist Jay Bernard: ‘I think our problem today is
        ↪  technical illiteracy’"
    },
    {
        "id": "682596f8cb540dcb19b74b66",
        "text": "‘It never happened - but the picture says it did’: 28 fake
        ↪  images that fooled the world ~"Pictures or it didn’t happen." So
        ↪  runs the immediate social media retort to any claim deemed too
        ↪  extraordinary to be true...",
        "newspaper": "GUARDIAN",
        "section": "api|GUARDIAN|Art and design",
        "date": "2025-04-12",
        "url": "https://www.theguardian.com/artanddesign/2025/apr/12/28-fake
        ↪  -images-that-fooled-the-world",
        "title": "‘It never happened - but the picture says it did’: 28 fake
        ↪  images that fooled the world"
    }
]
```


Listing 20: An example of getDocuments response

Another converted endpoint within the `tm` service is `/query/queryId`, which retrieves a query from MongoDB based on the provided `queryId`. This REST endpoint was also transitioned to gRPC and evaluated using k6 in a manner similar to the `getDocuments` method. However, the performance results did not provide any additional insights beyond those discussed in Section 4.3. For this reason, the conversion steps and performance outcomes for this endpoint are

omitted.

3.4.1 Conversion Steps

The following steps were conducted to refactor the REST method to gRPC for this scenario:

- **REST API Endpoint Identification**
  The endpoint converted is `/documents`. Listing 21 shows its implementation, while Listing 22 shows how the `api` service calls it.

  The `getDocuments` method exposed by the `tm` service is implemented as an HTTP-based API using the POST method, even though its primary purpose is to retrieve data. The request body contains parameters such as query ID, topic index, and start/end indices, which control the selection and pagination of results. Although the endpoint uses a resource-oriented path, it does not fully conform to REST principles because it does not leverage the proper semantics of HTTP methods—GET would be more appropriate for data retrieval. According to the Richardson Maturity Model (RMM) [10], this places the `getDocuments` method at Level 1, where resources are identified through URIs but HTTP verbs are not used consistently with REST conventions. The response also lacks hypermedia controls, which are required to reach Level 3. Therefore, while the method is commonly referred to as RESTful due to its use of HTTP and resource naming, it does not fully adhere to REST constraints. Future improvements could involve transitioning the method to use GET with query parameters and incorporating HATEOAS elements to advance toward higher levels.

```
1 @PostMapping(value = "/documents", produces =
  ↪ MediaType.APPLICATION_JSON_VALUE)
2 public List<Document> getDocuments(@RequestBody DocumentsRequest
  ↪ documentsRequest) {
3     List<Document> documents = new ArrayList<>();
4     if (documentsRequest.isValid()) {
5         QueryTopicDocs queryTopicDocs =
          ↪ mongoDB.FindQueryTopicDocs(documentsRequest.getQueryId(),
          ↪ documentsRequest.getTopicIndex());
6         if (queryTopicDocs != null) {
7             List<Pair<String, Double>> topDocuments =
              ↪ queryTopicDocs.getTopDocuments();
8             int startIndex = documentsRequest.getStartIndex();
9             int endIndex = documentsRequest.getEndIndex();
```

```
            if (startIndex < 0 || endIndex < startIndex)
                return new ArrayList<>();
            List<String> documentIds = new ArrayList<>();
            for (int i = startIndex; i < endIndex && i <
            ↪  topDocuments.size(); i++)
                documentIds.add(topDocuments.get(i).getL());
            documents = elasticClient.SearchDocuments(documentIds,
            ↪  queryTopicDocs.getElasticIndex());
        }
    }
    return documents;
}
```

Listing 21: REST implementation of getDocuments method

```
var options = {
    hostname: tips-tm,
    port: 8080,
    path: '/documents',
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    }
};

var f_req = http.request(options, function (f_res) {
    var chunks = '';

    f_res.setEncoding('utf8');

    f_res.on('data', function (chunk) {
        chunks += chunk;
    });

    f_res.on('end', function () {
        try {
            const object = JSON.parse(chunks);
            res.send(object)
```

```
24          } catch (error) {  res.status(500).json({success: false, msg:
            ↪  'Impossible to parse the list of documents'});
25          }
26      });
27  });
28  f_req.write(JSON.stringify(req.body));
29  f_req.end();
```

Listing 22: The api's call to getDocuments REST endpoint

- **gRPC Installation**

  For Node, the required dependencies are `@grpc/grpc-js` and `@grpc/proto-loader` [29]. Regarding tm service, implemented in Spring, the needed dependencies are `grpc-server-spring-boot-starter`, `grpc-netty-shaded`, `grpc-protobuf` and `protobuf-java` [30].

- **Protocol Buffers Definition**

  Listing 23 shows the definition of the gRPC method corresponding to the `getDocuments` REST endpoint, including the request and response message types.

```
1   syntax = "proto3";
2
3   package tips;
4
5   service TipsService {
6     rpc GetDocuments (GetDocumentsRequest) returns (GetDocumentsResponse);
7   }
8
9   message GetDocumentsRequest {
10    string queryId = 1;
11    int32 topicIndex = 2;
12    int32 startIndex = 3;
13    int32 endIndex = 4;
14  }
15
16  message Document {
17    string id = 1;
18    string text = 2;
19    string newspaper = 3;
20    string section = 4;
```

```
  string date = 5;
  string url = 6;
  string title = 7;
}

message GetDocumentsResponse {
  repeated Document documents = 1;
}
```

Listing 23: gRPC GetDocuments method definition

- **gRPC Code Generation**

  When it comes to Node, to load a `.proto` file use `loadSync()` method from `@grpc/proto-loader` library and pass the output to `loadPackageDefinition` method from `@grpc/grpc-js` library. The Node.js library dynamically generates service descriptors and client stub definitions from `.proto` files loaded at runtime [29]. Regarding tm-service, implemented in Spring where Maven is used, a build plugin can generate the necessary code as part of the build with `protobuf-maven-plugin` [31]. Listing 24 shows added code to `plugins` section inside `build` block which generates necessary code.

```
<plugin>
    <groupId>kr.motd.maven</groupId>
    <artifactId>os-maven-plugin</artifactId>
    <version>1.7.0</version>
    <executions>
        <execution>
            <goals>
                <goal>detect</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.6.1</version>
    <configuration>
       <protocArtifact>            com.google.protobuf:protoc:${protobu
        ↪ f-java.version}:exe:${os.detected.classifier}
        </protocArtifact>
```

```
20          <pluginId>grpc-java</pluginId>
21          <pluginArtifact>
22          io.grpc:protoc-gen-grpc-java:${grpc.version}:exe:${os.detected.
            ↪  classifier}
23          </pluginArtifact>
24      </configuration>
25      <executions>
26          <execution>
27              <id>compile</id>
28              <goals>
29                  <goal>compile</goal>
30                  <goal>compile-custom</goal>
31              </goals>
32              <configuration><pluginParameter>jakarta_omit,@generated=omi
                ↪  t</pluginParameter>
33              </configuration>
34          </execution>
35      </executions>
36  </plugin>
```

Listing 24: gRPC code generation for Spring

- **gRPC Method implementation**

  The next step is gRPC `getDocuments` method implementation which corresponds to the REST method shown in Listing 21. Listing 25 shows the gRPC version of the method.

```
1   @GrpcService
2   public class GrpcServerService extends
    ↪  TipsServiceGrpc.TipsServiceImplBase {
3
4       @Autowired
5       private MongoDB mongoDB;
6
7       @Autowired
8       private ElasticClient elasticClient;
9
10      @Override
11      public void getDocuments(GetDocumentsRequest request,
        ↪  StreamObserver<GetDocumentsResponse> responseObserver) {
12          List<Document> documents = new ArrayList<>();
```

```
13          int startIndex = request.getStartIndex();
14          int endIndex = request.getEndIndex();
15
16          if (request.getQueryId() != null) {
17              QueryTopicDocs queryTopicDocs =
                ↪  mongoDB.FindQueryTopicDocs(request.getQueryId(),
                ↪  request.getTopicIndex());
18              if (queryTopicDocs != null) {
19                  List<Pair<String, Double>> topDocuments =
                    ↪  queryTopicDocs.getTopDocuments();
20                  if (startIndex >= 0 && endIndex >= startIndex) {
21                      List<String> documentIds = new ArrayList<>();
22                      for (int i = startIndex; i < endIndex && i <
                        ↪  topDocuments.size(); i++) {
23                          documentIds.add(topDocuments.get(i).getL());
24                      }
25                      documents =
                        ↪  elasticClient.SearchDocumentsGrpc(documentIds,
                        ↪  queryTopicDocs.getElasticIndex());
26                  }
27              }
28          }
29
30          GetDocumentsResponse response =
            ↪  GetDocumentsResponse.newBuilder()
31                  .addAllDocuments(documents)
32                  .build();
33
34          responseObserver.onNext(response);
35          responseObserver.onCompleted();
36      }
37  }
```

Listing 25: gRPC getDocuments method implementation

- **gRPC Server creation**

  Creating a server consists of overriding the service base class generated from our service definition and running a gRPC server to listen for requests from clients and return the service responses [31]. @GrpcService annotation on `GrpcServerService` class marks it to be registered with a

gRPC server. By default, the gRPC server runs on 9090 port [30].

- **gRPC Client creation**

  After the necessary code has been dynamically generated, the stub constructor is in the `tips` namespace. In order to call service methods the stub has to be created. To do that, the `TipsService` stub constructor has to be called, specifying the server address and port [29]. All necessary steps for the client creation are presented at Listing 27.

```
var grpc = require("@grpc/grpc-js");
var protoLoader = require("@grpc/proto-loader");

const PROTO_PATH = path.resolve(__dirname,
↪  "../../../../proto/tips.proto");
var packageDefinition = protoLoader.loadSync(PROTO_PATH, {
  keepCase: true,
  longs: String,
  enums: String,
  defaults: true,
  oneofs: true,
});

let tipsProto = grpc.loadPackageDefinition(packageDefinition).tips;

const tipsClient = new tipsProto.TipsService(
  "tips-tm" + ":9090",
  grpc.credentials.createInsecure()
);
```

Listing 26: gRPC client setup for Node

```
const { queryId, topicIndex, startIndex, endIndex } = req.body;

const grpcRequest = {
  queryId: queryId,
  topicIndex: parseInt(topicIndex),
  startIndex: parseInt(startIndex),
  endIndex: parseInt(endIndex),
};

tipsClient.GetDocuments(grpcRequest, (error, response) => {
```

```
11      if (!response || !response.documents) {
12      return res.status(404).send("No documents found");
13      }
14
15      res.send(response.documents);
16  });
```

Listing 27: The api's call to getDocuments gRPC method

Chapter 4

Results and Discussion

4.1 Test Configuration

The performance tests were executed locally on a laptop. The machine used was an *HP EliteBook 840 14 inch G10 Notebook PC* with the following specifications: a 13th Gen Intel(R) Core(TM) i5-1345U processor running at 1.60 GHz, 128 MB Intel(R) UHD Graphics, and 16 GB of installed RAM operating at 5600 MHz. The system type is a 64-bit operating system on an x64-based processor, running the *Windows 11 Pro* edition. All Docker containers and services were run locally under this environment.

4.2 Collector - Processor Use Case

4.2.1 Test Setup

This section describes the setup used to compare the performance of the REST and gRPC servers within the Use Case explained in Section 3.3. The comparison is performed using Prometheus and Grafana.

Prometheus scrapes metrics from the following jobs: *grpc-server* and *rest-server*. A snippet of the `prometheus.yml` configuration file defining these jobs is shown in Listing 28. The `scrape_configs` section specifies which targets Prometheus should monitor—in this case, the *grpc-server* and *rest-server* services.

```
1 scrape_configs:
2 - job_name: 'grpc-server'
3   static_configs:
4     - targets: ['tips-grpc-server:8050']
5
```

```
6  - job_name: 'rest-server'
7    static_configs:
8      - targets: ['tips-rest-server:8000']
```

Listing 28: Prometheus job configuration

In Listings 11 and 15, line 7 shows the command used to start the Prometheus metrics server on ports 8000 and 8050 for the REST and gRPC servers, respectively. Additionally, three metrics are used to compare the performance of both servers: REQUEST_COUNT, REQUEST_LATENCY, and FAIL_COUNT. These metrics are initialized between lines 3 and 6 in the respective listings. The REQUEST_COUNT metric records the number of requests to the NER extraction method on each server. REQUEST_LATENCY measures the processing time for each request, while FAIL_COUNT tracks the number of failed requests.

Grafana uses Prometheus as its data source. The configuration for this setup is provided in Listing 29.

```
1  datasources:
2  - name: Prometheus
3    type: prometheus
4    url: http://prometheus-server:9090
```

Listing 29: Grafana data source configuration

A Grafana dashboard titled *gRPC vs REST Processor Servers Monitoring* was created to visualize the metrics and compare the servers. This dashboard includes the following panels: Average Latencies, gRPC Avg Latency, REST Avg Latency, gRPC Failed Requests and REST Failed Requests.

The following test scenarios were designed to compare the two servers:

- NER extraction of 200 documents
- NER extraction of 500 documents
- NER extraction of 1000 documents
- NER extraction of 1500 documents
- NER extraction of 2000 documents
- Sending 2000 documents without performing extraction

In each scenario, documents from the past 24 hours are fetched from the Guardian API and sent individually from the *collector* service to the *processor* service for extraction. To ensure result accuracy, each test is executed four times for both the REST and gRPC servers.

4.2.2 Test Results

As it is mentioned in Section 4.2.1 six different test scenarios were conducted to analyze the requests. In this section, the results of each test scenario are shown and discussed. It is important to note that each test scenario was executed sequentially—first the REST method, followed by the gRPC method—and this process was repeated four times per scenario. In other words, there was no parallel execution, and gRPC tests only began after the corresponding REST tests had completed. As a result, the graphs presented in Section 4.2.2 are not time-aligned across the two protocols.

NER Extraction of 200 Documents

In this performance test scenario, both REST and gRPC servers were evaluated for the NER process over 200 documents. The test was executed in four iterations per protocol, resulting in a total of 800 requests per server. The total time for each iteration was recorded and is summarized in Table 4.1. Additionally, the average latency for each protocol was monitored and visualized using Grafana dashboards.

Table 4.1: Total time per iteration for NER extraction (200 documents)

Iteration	REST (s)	gRPC (s)
1.	25.20	21.35
2.	18.31	18.27
3.	18.70	17.85
4.	18.68	18.42

As seen in Table 4.1, the REST protocol showed a higher response time during the first iteration (25.20s) compared to gRPC (21.35s). In the remaining iterations, both protocols demonstrated more consistent performance. The average total processing times over four iterations were:

- REST: $\frac{25.20+18.31+18.70+18.68}{4} = 20.22\,s$
- gRPC: $\frac{21.35+18.27+17.85+18.42}{4} = 18.97\,s$

On average, gRPC was faster by approximately 1.25 seconds per iteration.
Figure 4.1 displays the average latency for both protocols, as measured by Prometheus and visualized through Grafana. The gRPC average latency was 93.69 ms, while REST exhibited a slightly higher average latency of 98.19 ms.

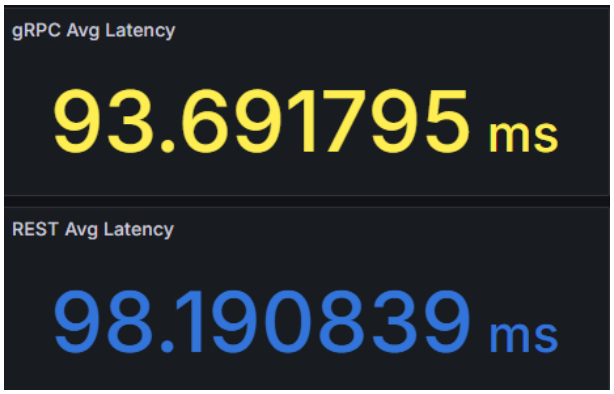

Figure 4.1: Average latency of gRPC and REST (200 documents)

Figure 4.2 provides a more detailed view of latency over time. It shows that gRPC consistently maintains lower latency than REST throughout most of the test window—as mentioned earlier, the starting point of the yellow line should be aligned with that of the blue line. Initial fluctuations in latency are evident for both protocols, but gRPC stabilizes more quickly and achieves lower latency values in the steady state.

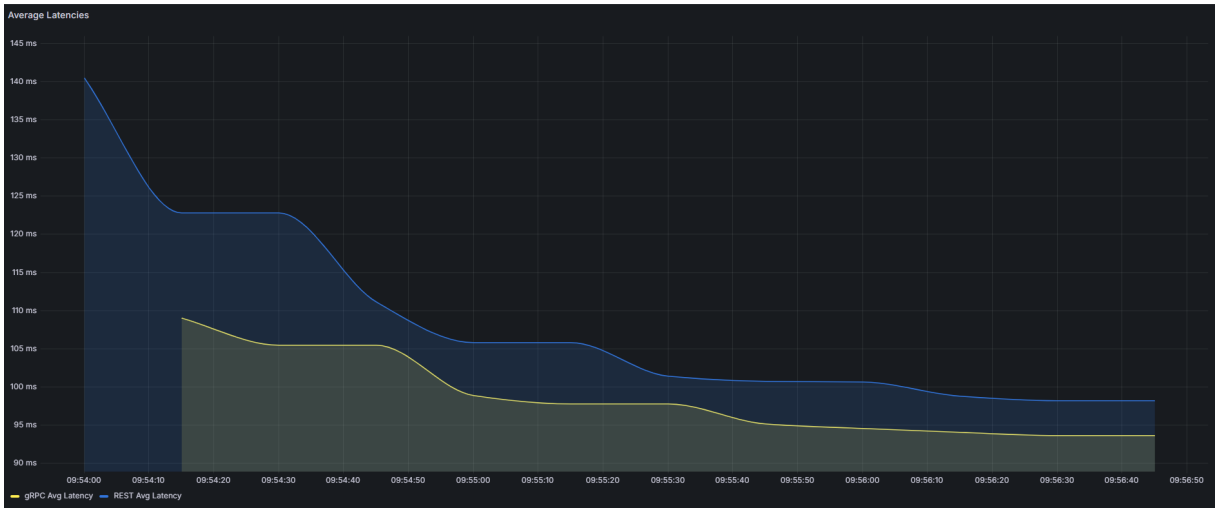

Figure 4.2: Latency trends over time for REST and gRPC (200 documents)

These results demonstrate that gRPC provides better performance for high-volume NER requests. It not only completes the workload faster but also maintains lower and more consistent latency.

NER Extraction of 500 Documents

This scenario involved extracting named entities from 500 documents per iteration. As shown in Table 4.2, gRPC generally demonstrated better performance, completing most iterations faster than REST. However, in the third iteration, REST slightly outperformed gRPC, highlighting

some variability in response times. Despite this, the overall trend still favors gRPC, as reflected in the average latency and latency trend graphs (Figures 4.3 and 4.4).

Table 4.2: Total time per iteration for NER extraction (500 documents)

Iteration	REST (s)	gRPC (s)
1.	52.97	51.98
2.	49.5	46.41
3.	47.82	48.48
4.	49	47.67

Figure 4.3: Average latency of gRPC and REST (500 documents)

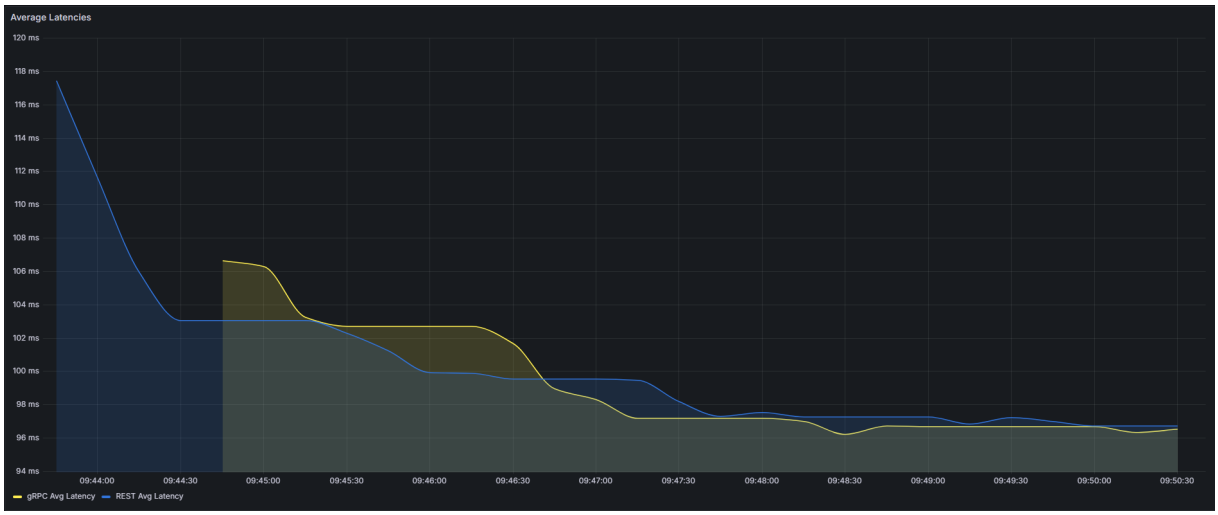

Figure 4.4: Latency trends over time for REST and gRPC (500 documents)

NER Extraction of 1000 Documents

When the number of documents was increased to 1000, the difference between the two protocols became more noticeable. As displayed in Table 4.3, gRPC completed the processing faster in

each iteration. The latency figures further support gRPC's advantage, with both the average latency and latency trend graphs (Figures 4.5 and 4.6) reflecting lower values compared to REST.

Table 4.3: Total time per iteration for NER extraction (1000 documents)

Iteration	REST (s)	gRPC (s)
1.	105.79	102.52
2.	98.03	94.75
3.	97.86	94.26
4.	97.16	93.99

Figure 4.5: Average latency of gRPC and REST (1000 documents)

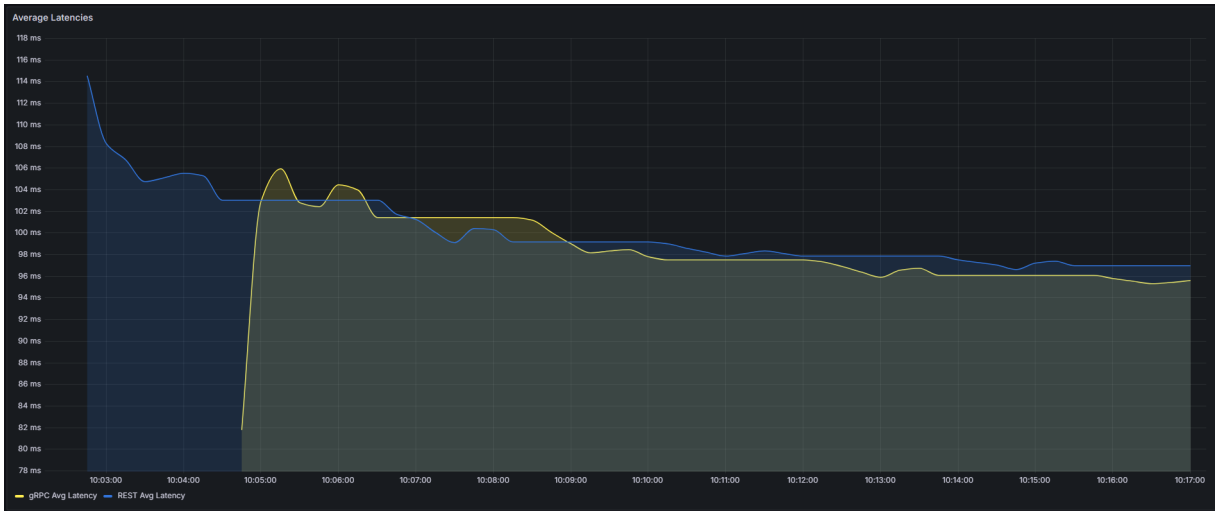

Figure 4.6: Latency trends over time for REST and gRPC (1000 documents)

NER Extraction of 1500 Documents

The 1500-document scenario introduced a heavier load. While REST experienced a sharp spike in response time during the first iteration, gRPC remained consistent across all runs (Table 4.4).

The latency metrics (Figures 4.7 and 4.8) reinforce gRPC's superior stability and responsiveness under increased load.

Table 4.4: Total time per iteration for NER extraction (1500 documents)

Iteration	REST (s)	gRPC (s)
1.	267.41	152.16
2.	146.23	140.23
3.	145.42	141.64
4.	143.44	140.38

Figure 4.7: Average latency of gRPC and REST (1500 documents)

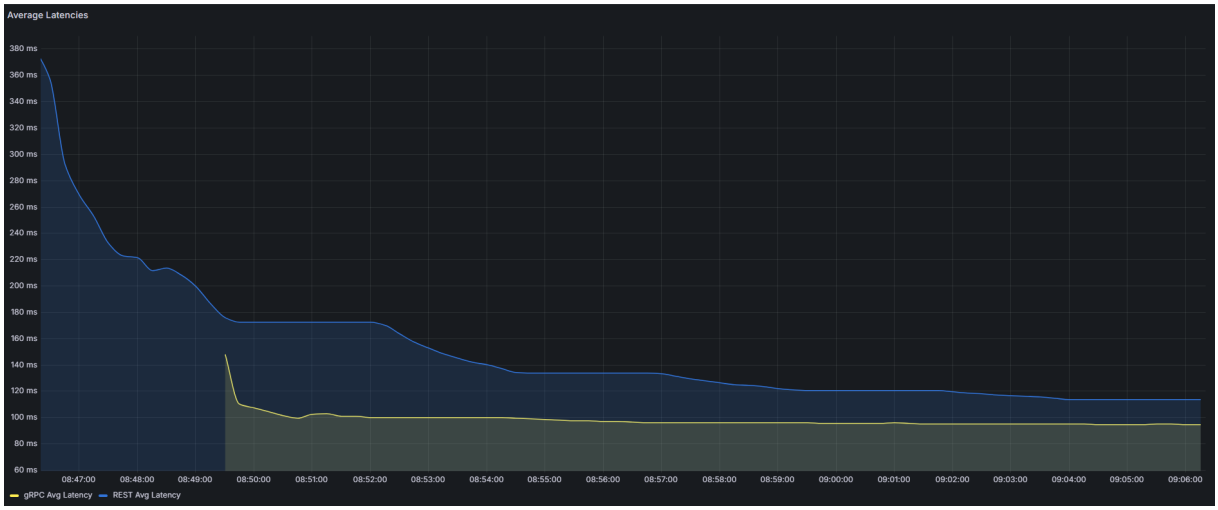

Figure 4.8: Latency trends over time for REST and gRPC (1500 documents)

NER Extraction of 2000 Documents

As the document count reached 2000, the trends observed in previous tests continued. gRPC showed lower execution times across the board, outperforming REST in every iteration (Ta-

ble 4.5). The latency graphs (Figures 4.9 and 4.10) again indicate that gRPC maintained more consistent and lower latency than REST under heavier loads.

Table 4.5: Total time per iteration for NER extraction (2000 documents)

Iteration	REST (s)	gRPC (s)
1.	201.26	196.71
2.	193.25	187.8
3.	193.66	182.86
4.	196.26	183.98

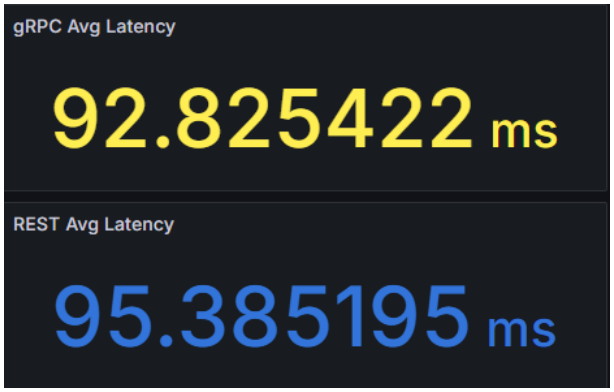

Figure 4.9: Average latency of gRPC and REST (2000 documents)

Figure 4.10: Latency trends over time for REST and gRPC (2000 documents)

Sending 2000 documents without performing extraction

To isolate the overhead of communication alone, this scenario measured the time taken to send 2000 documents without performing NER. As expected, both protocols completed the task more

quickly, but gRPC achieved lower times, averaging below 1 second per iteration (Table 4.6). Figures 4.11 and 4.12 clearly illustrate that gRPC is far more efficient in pure data transmission scenarios.

Table 4.6: Total time per iteration without NER (2000 documents)

Iteration	REST (s)	gRPC (s)
1.	3.46	0.87
2.	3.45	0.87
3.	3.69	0.89
4.	4.28	0.9

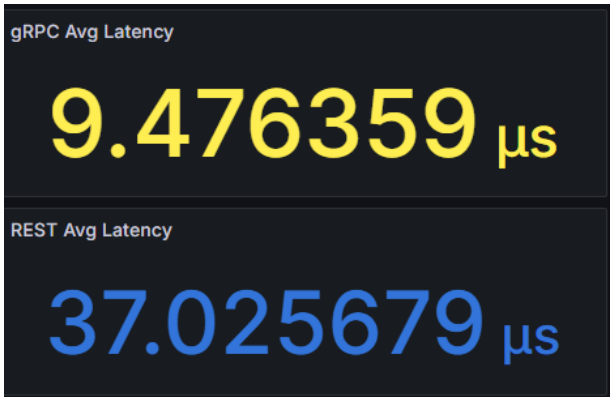

Figure 4.11: Average latency of gRPC and REST (2000 documents without NER)

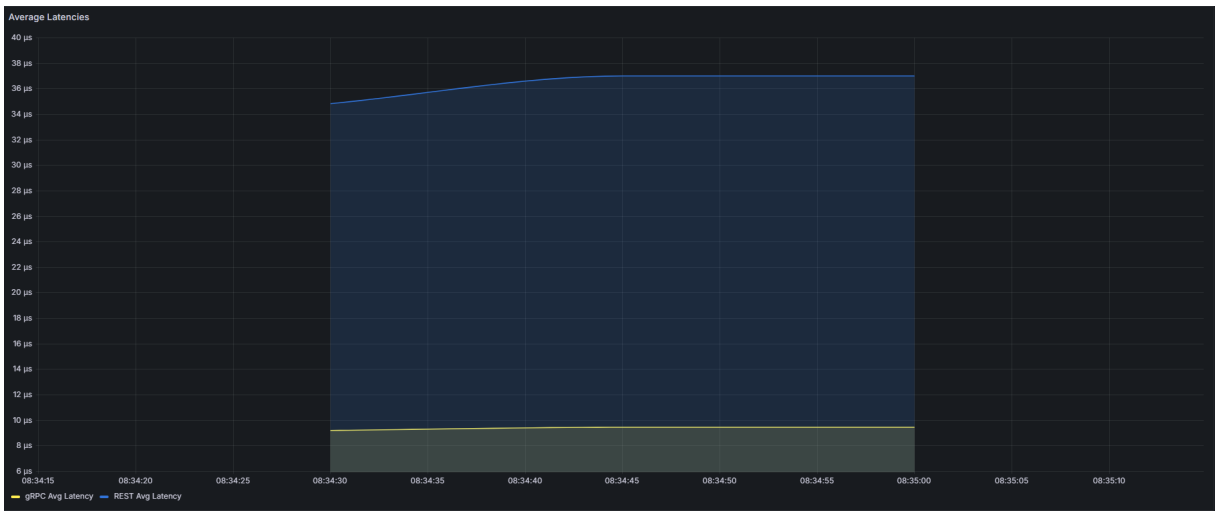

Figure 4.12: Latency trends over time for REST and gRPC (2000 documents without NER)

4.2.3 Test Results Discussion

Across all test scenarios, gRPC consistently demonstrated better performance than REST in terms of both total processing time and latency. While the performance gap varied across tests, gRPC generally achieved faster iteration times and maintained lower average latency, particularly as the document volume increased. However, it is important to note that in most scenarios, the time taken to perform the NER extraction constituted the majority of the total execution time. As such, the differences between the communication protocols were partially masked by the computational cost of processing the documents. This changed in the final test scenario, where 2000 documents were transmitted without performing NER. With the computational overhead removed, the performance advantage of gRPC over REST became much more pronounced—highlighting the efficiency of the protocol itself. This suggests that while both protocols are suitable for high-load NER tasks, gRPC offers benefits in raw transmission performance, especially in scenarios where communication speed is critical.

4.3 API - TM Use Case

4.3.1 Test Setup

This section describes the setup used to compare the performance of the REST and gRPC protocols for the use case described in Section 3.4. The comparison was performed using `k6`, the load testing tool explained in Section 2.1.5. Four separate `k6` test scripts were created: `test_rest_direct.js`, `test_grpc_direct.js`, `test_rest_via_api.js`, and `test_grpc_via_api.js`. The first two were used to evaluate REST and gRPC methods directly within the `tm` service, while the latter two tested REST and gRPC endpoints exposed through the `api` service.

This two-level testing strategy was designed to verify whether the `api` service acts as a performance bottleneck. If one protocol is consistently faster when accessed directly, it is expected to maintain this advantage when accessed via the `api` gateway.

A `sleep(0.1)` delay follows each request to prevent overwhelming the server and to ensure consistent pacing between requests.

In all four scripts, the `topicIndex` value is generated dynamically depending on the number of virtual users (VUs). If only one VU is used, the `topicIndex` changes with each iteration to avoid sending identical payloads. When more than one VU is involved, the `topicIndex` is derived from the VU number. This variation prevents server-side caching and helps simulate more realistic usage patterns.

Listings 30 to 33 show the contents of each `k6` test script.

```javascript
import http from "k6/http";
import { check, sleep } from "k6";

export const options = {
  vus: __ENV.VUS,
  iterations: __ENV.ITERATIONS
};

const REST_URL = "http://localhost:8080/documents";

const headers = {
  "Content-Type": "application/json",
};

export default function () {
  let topicIndex;
  if (__ENV.VUS == 1) {
    topicIndex = __ITER % 11;
  } else {
    topicIndex = (__VU - 1) % 11;
  }

  const payload = {
    queryId: "6835aab0c8dd352b32858a56",
    topicIndex: topicIndex,
    startIndex: 0,
    endIndex: 50,
  };

  const restRes = http.post(REST_URL, JSON.stringify(payload), { headers });

  sleep(0.1);
}
```

Listing 30: k6 script for testing REST directly

```javascript
import { check, sleep } from "k6";
import grpc from "k6/net/grpc";
```

```
3
4  export const options = {
5    vus: __ENV.VUS,
6    iterations: __ENV.ITERATIONS
7  };
8
9  const GRPC_HOST = "localhost:9090";
10
11 const client = new grpc.Client();
12 client.load(["."], "../src/main/proto/tips.proto");
13
14 export default function () {
15   let topicIndex;
16   if (__ENV.VUS == 1) {
17     topicIndex = __ITER % 11;
18   } else {
19     topicIndex = (__VU - 1) % 11;
20   }
21
22   const payload = {
23     queryId: "6835aab0c8dd352b32858a56",
24     topicIndex: topicIndex,
25     startIndex: 0,
26     endIndex: 50,
27   };
28
29   client.connect(GRPC_HOST, { plaintext: true });
30   const grpcRes = client.invoke("tips.TipsService/GetDocuments", payload);
31   client.close();
32
33   sleep(0.1);
34 }
```

Listing 31: k6 script for testing gRPC directly

```
1  import http from "k6/http";
2  import { check, sleep } from "k6";
3
4  export const options = {
```

```
  vus: __ENV.VUS,
  iterations: __ENV.ITERATIONS,
};

const BASE_URL = "http://localhost:3010/api/v1";
const AUTH_HEADER =
  "JWT...";

const headers = {
  "Content-Type": "application/json",
  Authorization: AUTH_HEADER,
};

export default function () {
  let topicIndex;
  if (__ENV.VUS == 1) {
    topicIndex = __ITER % 11;
  } else {
    topicIndex = (__VU - 1) % 11;
  }

  const payload = {
    queryId: "6835aab0c8dd352b32858a56",
    topicIndex: topicIndex,
    startIndex: 0,
    endIndex: 50,
  };

  const restRes = http.post(${BASE_URL}/documents, JSON.stringify(payload), {
    headers,
  });

  sleep(0.1);
}
```

Listing 32: k6 script for testing REST via `api`

```
import http from "k6/http";
import { check, sleep } from "k6";
```

```
3
4  export const options = {
5    vus: __ENV.VUS,
6    iterations: __ENV.ITERATIONS,
7  };
8
9  const BASE_URL = "http://localhost:3010/api/v1";
10 const AUTH_HEADER =
11   "JWT...";
12
13 const headers = {
14   "Content-Type": "application/json",
15   Authorization: AUTH_HEADER,
16 };
17
18 export default function () {
19   let topicIndex;
20   if (__ENV.VUS == 1) {
21     topicIndex = __ITER % 11;
22   } else {
23     topicIndex = (__VU - 1) % 11;
24   }
25
26   const payload = {
27     queryId: "6835aab0c8dd352b32858a56",
28     topicIndex: topicIndex,
29     startIndex: 0,
30     endIndex: 50,
31   };
32
33   const grpcRes = http.post(
34     ${BASE_URL}/grpc/documents,
35     JSON.stringify(payload),
36     { headers }
37   );
38
39   sleep(0.1);
40 }
```

Listing 33: k6 script for testing gRPC via `api`

The `k6` command used to execute one of the test scripts is shown in Listing 34. Depending on the specific test scenario, the script name should be adjusted accordingly. The values for the environment variables `VUS` and `ITERATIONS` are also modified to match each test case.

```
k6 --env VUS=1 --env ITERATIONS=500 run .\test_rest_direct.js
```

Listing 34: `k6` run command for testing REST directly

The following test scenarios were executed to evaluate and compare the two protocols:

- `getDocuments` method with 1 VU and 200 iterations, returning only an empty array without executing the underlying logic
- `getDocuments` method with 1 VU and 200 iterations
- `getDocuments` method with 1 VU and 500 iterations
- `getDocuments` method with 10 VUs and 500 iterations
- `getDocuments` method with 50 VUs and 500 iterations
- `getDocuments` method with 50 VUs and 5000 iterations

These test scenarios differ from those used in the `collector-processor` use case, as the nature of the method invocation is different. While the method in the `collector-processor` use case typically runs once per day over approximately 1500 documents, the `getDocuments` method in the `api-tm` use case can be triggered on-demand by multiple users simultaneously or by a single user several times throughout the day. Therefore, the `api-tm` use case places different performance demands on the system.

4.3.2 Test Results

This section presents the results of the test scenarios described in Section 4.3.1.

getDocuments method with 1 VU and 200 iterations, returning only an empty array without executing the underlying logic

In this scenario, the `getDocuments` method was tested directly within the `tm` service, omitting any internal logic. Both protocols returned empty arrays. The results in Table 4.7 show that gRPC outperformed REST.

Table 4.7: Summary of Results: Direct gRPC vs REST (no logic, 1 VU, 200 iterations)

Metric	**REST**	**gRPC**
Average Duration (ms)	6.75	3.62
Failed Requests	0%	0%

Table 4.8 shows results when the same request was sent via the `api` gateway. While gRPC remained faster, both protocols experienced slightly higher durations due to the additional API layer.

Table 4.8: Summary of Results: gRPC vs REST via API (no logic, 1 VU, 200 iterations)

Protocol	**REST**	**gRPC**
Average Duration (ms)	13.22	12.31
Failed Requests	0%	0%

getDocuments method with 1 VU and 200 iterations

Under a light load, gRPC consistently outperformed REST in direct calls, with a noticeable difference in average duration. However, when routed through the `api` layer, both protocols performed similarly.

Table 4.9: Summary of Results: Direct gRPC vs REST (1 VU, 200 iterations)

Metric	**REST**	**gRPC**
Average Duration (ms)	27.48	21.96
Failed Requests	0%	0%

Table 4.10: Average Duration (ms): gRPC vs REST via API (1 VU, 200 iterations)

Protocol	**REST**	**gRPC**
Average Duration (ms)	30.38	30.29

getDocuments method with 1 VU and 500 iterations

In this test scenario, gRPC remained slightly faster in direct comparisons. However, the performance gap disappeared when accessed via the API, where gRPC was slower than REST.

Table 4.11: Summary of Results: Direct gRPC vs REST (1 VU, 500 iterations)

Metric	**REST**	**gRPC**
Average Duration (ms)	20.44	18.34
Failed Requests	0%	0%

Table 4.12: Average Duration (ms): gRPC vs REST via API (1 VU, 500 iterations)

Protocol	**REST**	**gRPC**
Average Duration (ms)	26.24	27.61

getDocuments method with 10 VUs and 500 iterations

As load increased to 10 virtual users, REST began to outperform gRPC in both direct and API-routed scenarios.

Table 4.13: Summary of Results: Direct gRPC vs REST (10 VUs, 500 iterations)

Metric	**REST**	**gRPC**
Average Duration (ms)	25.9	29.69
Failed Requests	0%	0%

Table 4.14: Average Duration (ms): gRPC vs REST via API (10 VUs, 500 iterations)

Protocol	**REST**	**gRPC**
Average Duration (ms)	31.4	33.5

getDocuments method with 50 VUs and 500 iterations

Under heavier load (50 VUs), the performance gap between protocols narrowed. Direct calls slightly favored gRPC, but REST clearly outperformed gRPC when routed via the API.

Table 4.15: Summary of Results: Direct gRPC vs REST (50 VUs, 500 iterations)

Metric	**REST**	**gRPC**
Average Duration (ms)	39.4	38.5
Failed Requests	0%	0%

Table 4.16: Average Duration (ms): gRPC vs REST via API (50 VUs, 500 iterations)

Protocol	**REST**	**gRPC**
Average Duration (ms)	146.3	169.9

getDocuments method with 50 VUs and 5000 iterations

At very high volume and concurrency, REST outperformed gRPC in both direct and API-based calls.

Table 4.17: Summary of Results: Direct gRPC vs REST (50 VUs, 5000 iterations)

Metric	REST	gRPC
Average Duration (ms)	53.3	50.58
Failed Requests	0%	0%

Table 4.18: Average Duration (ms): gRPC vs REST via API (50 VUs, 5000 iterations)

Protocol	REST	gRPC
Average Duration (ms)	232.2	279.5

4.3.3 Test Results Discussion

Figure 4.13 provides a visual summary of the average response times across all test scenarios, comparing gRPC and REST both in direct communication and via the `api` gateway.

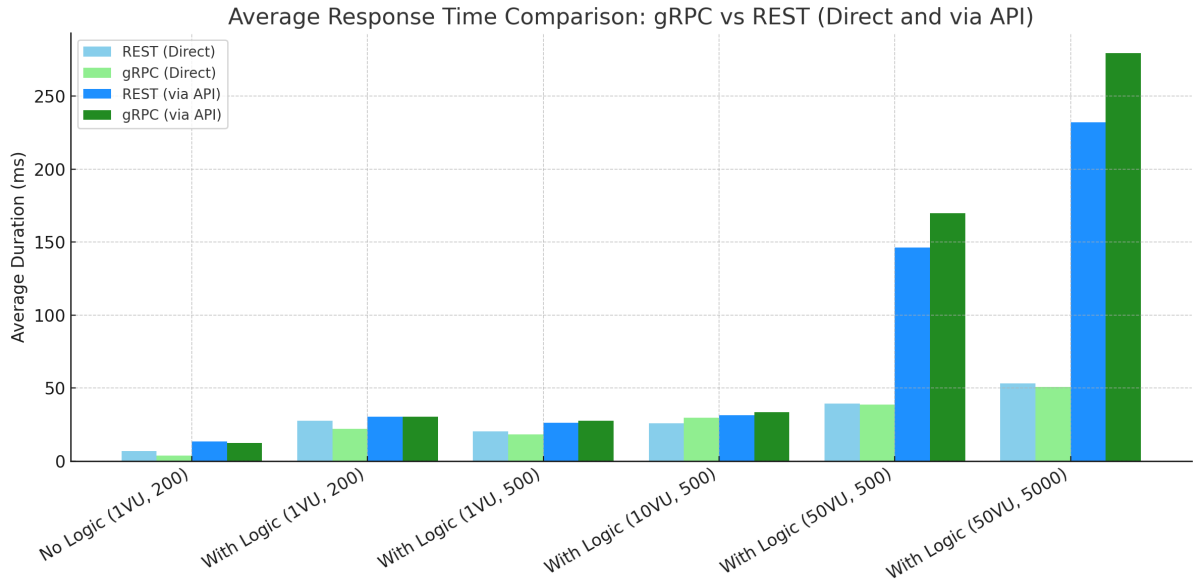

Figure 4.13: Comparison of average response times between gRPC and REST (direct vs. via API)

The graph highlights several key trends. Under light load (1 VU, 200–500 iterations), gRPC consistently outperformed REST in direct communication. The differences were especially noticeable in the scenarios with no internal logic, where gRPC achieved nearly half the average response time of REST. Even when routed through the `api`, gRPC generally remained faster or comparable, although the performance gap diminished due to the overhead introduced by the additional gateway layer.

However, as concurrency and volume increased (e.g., 50 VUs and 500 or 5000 iterations), the gap between REST and gRPC narrowed. In some of the high-load scenarios involving the `api` layer, REST outperformed gRPC.

Overall, gRPC demonstrated superior performance under typical and moderate workload. REST, however, was faster under high-load conditions when routed through the gateway.

Chapter 5

Conclusion

This thesis examined the performance implications of inter-service communication in microservice architectures by comparing two widely used architectural styles: REST and gRPC. The study was conducted using the TIPS application, a real-world system developed for educational and research purposes. Two use cases were analyzed: communication between the `collector` and `processor` services, and between the `api` and `tm` services.
The migration from REST to gRPC was implemented and tested for both use cases, and a range of load testing scenarios was conducted to measure and compare the performance of each protocol. Prometheus and Grafana were used for monitoring and observability in one use case, while k6 was used as a load testing tool in the other.

5.1 Key Findings

The findings from both setups are summarized below.

- **gRPC outperforms REST in low-load scenarios:** In both use cases, gRPC consistently achieved lower average response times than REST when tested with 1 virtual user (VU). This was particularly evident when services bypassed any heavy internal logic, revealing the raw communication efficiency of gRPC.
- **Python-based use case showed clear gRPC advantage:** In the first use case, where both services were implemented in Python, gRPC outperformed REST across all test scenarios. Even as document volume increased, gRPC maintained faster completion times. The greatest difference was observed when NER processing was skipped entirely, confirming the transmission efficiency of gRPC.
- **API gateway adds noticeable overhead:** In the second use case, introducing the `api` gateway increased response times for both protocols. While gRPC was still faster in many

scenarios, the difference became smaller or even reversed in some high-concurrency situations.

- **REST scales better under high load through the API:** Under high-load conditions (e.g., 50 VUs and 5000 iterations), REST outperformed gRPC when requests were routed via the API gateway.
- **Protocol choice depends on use case characteristics:** gRPC is well-suited for internal service-to-service communication where low latency and efficiency are critical. REST, on the other hand, remains a strong choice for publicly exposed APIs or scenarios with heavier routing and high concurrency.

These results suggest that gRPC is a performant and efficient protocol for internal microservice communication, especially when latency is a priority. REST continues to be a reliable option, particularly when serving external clients or handling high-load traffic via API gateways.

5.2 Contributions

The main contributions of the thesis are listed below.

- A complete, real-world implementation of a protocol migration from REST to gRPC within a microservice-based system.
- A systematic performance evaluation using a combination of observability tools (Prometheus and Grafana) and load testing (k6) across multiple scenarios.
- A set of recommendations on when gRPC should be preferred over REST, supported by empirical results.

5.3 Future Work

While this thesis focused on synchronous communication, future research could explore asynchronous communication using message queues (e.g., RabbitMQ) to analyze how event-driven architectures affect performance, scalability, and resilience.
Other potential directions include:

- Extending the number of services under test to assess communication complexity in larger-scale microservice systems.

- Exploring the use of load balancing techniques to enhance the performance and scalability of inter-service communication under varying workloads.
- Migrating certain microservices to alternative frameworks to evaluate how framework-level differences influence the performance.

In conclusion, the results support the claim that gRPC can improve performance in certain microservice communication scenarios. However, architectural decisions must still be based on the specific needs, infrastructure, and constraints of each system.

Bibliography

[1] M. Fowler and J. Lewis, “Microservices”, 2014, Accessed: 2025-05-20. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[2] A. Raikwar, *A beginner's guide to grpc*, Accessed: 2025-06-11, 2023. [Online]. Available: https://amitraikwar.medium.com/a-beginners-guide-to-grpc-2f580770b790.

[3] Grafana Labs, *Grafana*, Accessed: 2025-06-12, 2025. [Online]. Available: https://grafana.com/.

[4] E. Di Buccio, A. Cammozzo, F. Neresini, and A. Zanatta, “Tips: Search and analytics for social science research”, in *Proceedings of the 2nd Joint Conference of the Information Retrieval Communities in Europe (CIRCLE 2022)*, CEUR Workshop Proceedings, 2022. [Online]. Available: https://ceur-ws.org/Vol-3178/CIRCLE_2022_paper_33.pdf.

[5] Grafana Labs, *Grafana k6 documentation*, Accessed: 2025-06-20. [Online]. Available: https://grafana.com/docs/k6/latest/.

[6] Grafana Labs, *Types of load testing*, Accessed: 2025-05-21, 2024. [Online]. Available: https://grafana.com/load-testing/types-of-load-testing/.

[7] R. T. Fielding, “REST: architectural styles and the design of network-based software architectures”, Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[8] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture”, *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, 2002, issn: 1533-5399. doi: 10.1145/514183.514185. [Online]. Available: https://doi.org/10.1145/514183.514185.

[9] R. T. Fielding, R. N. Taylor, J. R. Erenkrantz, *et al.*, “Reflections on the rest architectural style and ”principled design of the modern web architecture” (impact paper award)”, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 4–14, isbn: 9781450351058. doi: 10.1145/3106237.3121282.

[10] M. Fowler, *Richardson maturity model*, https://martinfowler.com/articles/richardsonMaturityModel.html, Accessed: 2025-06-26, 2010.

[11] L. Gupta, *Restful api design*, Accessed: 2025-06-02, 2025. [Online]. Available: https://restfulapi.net/.

[12] *Grpc core concepts*, Accessed: 2025-06-03, 2024. [Online]. Available: https://grpc.io/docs/what-is-grpc/core-concepts/.

[13] *Performance best practices*, Accessed: 2025-06-11, 2024. [Online]. Available: https://grpc.io/docs/guides/performance/.

[14] Amazon Web Services, *The difference between grpc and rest*, Accessed: 2025-06-03, 2025. [Online]. Available: https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/.

[15] N. Siapno, *Performance testing vs. load testing vs. stress testing*, Accessed: 2025-06-12, 2023. [Online]. Available: https://blog.postman.com/performance-testing-vs-load-testing-vs-stress-testing/.

[16] N. Pottekkat, *An introduction to monitoring microservices*, Accessed: 2025-06-12, 2022. [Online]. Available: https://navendu.me/posts/introduction-to-monitoring-microservices/.

[17] *Prometheus overview*, Accessed: 2025-06-12, 2025. [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[18] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?", in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 138–149, isbn: 9781450369916. doi: 10.1145/3358960.3379124. [Online]. Available: https://doi.org/10.1145/3358960.3379124.

[19] V. Basavegowda Ramu, "Performance impact of microservices architecture", *The Review of Contemporary Scientific and Academic Studies*, vol. 3, 2023. doi: 10.55454/rcsas.3.06.2023.010.

[20] I. Ghani, W. M. N. Wan Kadir, A. Mustafa, and M. Babir, "Microservice testing approaches: A systematic literature review", vol. 11, pp. 75–80, Jan. 2019. doi: 10.30880/ijie.2019.11.08.008.

[21] H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, *Development frameworks for microservice-based applications: Evaluation and comparison*, Accessed: 2025-03-26, 2022. arXiv: 2203.07267 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2203.07267.

[22] R. Bux and G. S. Shenoy, "Performance analysis of restful web services and rabbitmq for microservices based systems on cloud environment", in *2024 3rd International Conference for Innovation in Technology (INOCON)*, 2024, pp. 1–6. doi: 10.1109/INOCON60754.2024.10511747.

[23] Y. Lee and Y. Liu, "Using refactoring to migrate rest applications to grpc", in *Proceedings of the 2022 ACM Southeast Conference*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 219–223, isbn: 9781450386975. doi: 10.1145/3476883.3520220. [Online]. Available: https://doi.org/10.1145/3476883.3520220.

[24] M. Rezaei and D. Subramani, *Migrating apis from rest to grpc at wepay*, Accessed: 2025-06-04, 2019. [Online]. Available: https://wecode.wepay.com/posts/migrating-apis-from-rest-to-grpc-at-wepay.

[25] R. Fernando, *Evaluating performance of rest vs grpc*, Accessed: 2025-06-03, 2019. [Online]. Available: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da.

[26] *Docker compose*, Accessed: 2025-06-04, 2024. [Online]. Available: https://docs.docker.com/compose/.

[27] *Grpc python quick start*, Accessed: 2025-06-06, 2024. [Online]. Available: https://grpc.io/docs/languages/python/.

[28] Y. Lee and Y. Liu, "Using refactoring to migrate rest applications to grpc", in *Proceedings of the 2022 ACM Southeast Conference*, New York, NY, USA: Association for Computing Machinery, 2022, isbn: 9781450386975. doi: 10.1145/3476883.3520220. [Online]. Available: https://doi.org/10.1145/3476883.3520220.

[29] *Grpc python quick start*, Accessed: 2025-06-17, 2024. [Online]. Available: https://grpc.io/docs/languages/node/.

[30] *Spring grpc*, Accessed: 2025-06-17. [Online]. Available: https://docs.spring.io/spring-grpc/reference/getting-started.html.

[31] *Grpc python quick start*, Accessed: 2025-06-17, 2024. [Online]. Available: https://grpc.io/docs/languages/java/.