jamk

A Comparative Study of REST, GraphQL, and gRPC for API Development

Practical Benchmarking with a Trello-Like Task Management System

Sven Reichersdörfer

Bachelor's thesis
December 2025
Degree Programme in Electrical and Automation Engineering

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Sven Reichersdörfer

A Comparative Study of REST, GraphQL, and gRPC for API Development
Practical Benchmarking with a Trello-Like Task Management System

Jyväskylä: Jamk University of Applied Sciences, December 2025, 89 pages

Degree Programme in Electrical Automation Engineering, Bachelor's Thesis

Permission for open access publication: Yes

Language of publication: English

Abstract

State-of-the-art software systems increasingly depend on Application Programming Interfaces (APIs) to enable interaction across distributed components and external services. Although REST has long been the dominant architectural style, alternative approaches such as GraphQL and gRPC have gained popularity due to their potential efficiency and flexibility advantages. The objective of the work was to compare these three API styles to identify their strengths and weaknesses in practical software development.

A task management application resembling Trello was implemented. The backend was developed using ASP.NET Core and SQLite, while the frontend was created with Blazor Server. The application supports the same functionality for the three APIs REST, GraphQL, and gRPC. The APIs were evaluated based on performance, developer experience, maintainability, and suitability for real-world use.

The evaluation showed that REST provided the most straightforward implementation and wide compatibility, GraphQL offered flexible data retrieval but added complexity to the setup, and gRPC achieved high performance and efficiency in communication but poses more challenges in debugging and the onboarding process.

Beyond performance metrics, this thesis provides a decision-making framework for developers, with specific criteria enabling them to select the optimal API architecture for their individual project constraints.

It was concluded that the most suitable API style depends on the context and requirements of the project. REST remains a reliable general-purpose choice, GraphQL is advantageous for data-driven applications with variable query needs, and gRPC is best suited for high-performance, service-to-service communication.

Keywords/tags (subjects)

API development, REST, GraphQL, gRPC, performance benchmarking, Blazor Server, ASP.NET Core, SQLite, Entity Framework Core, software architecture, developer experience, maintainability

Miscellaneous (Confidential information)

Contents

Figures

Tables

1 Introduction and Objectives

1.1 Motivation for the thesis

APIs act as invisible bridges that allow different software systems to communicate, playing a crucial role in ensuring the smooth operation of modern web and mobile applications (Software, 2024). The average application nowadays uses up to 50 APIs. This makes APIs the foundation of the development of modern applications (Postman, State of the API Report, 2024). Loganathan (2024) describes APIs as: “the modern equivalent of “plug and play” connectors in software.” This means that they streamline system-to-system communication and provide greater flexibility and autonomy across the enterprise.

REST is by far the most used API architecture but GraphQL and gRPC are increasingly used by developers (Postman, State of the API Report, 2025). Choosing the right API for a project is crucial as a developer. The three different API technologies REST, GraphQL and gRPC are widely used to build scalable, efficient and reliable APIs (Ahmad, 2025). Each API offers its own set of strengths and limitations. Selecting the wrong API for an application can lead to wasted development time – either implementing unnecessary functionality, dealing with sophisticated API infrastructure or adapting features that do not align with the project’s requirements.

1.2 Problem definition

As described in the preceding chapter, APIs play an important role in most modern applications. Developers often still face uncertainty about which API is best suited for their application and there is only scarce guidance on when to choose REST, GraphQL, or gRPC. Furthermore, Developers often default to REST even when another style may be more efficient (Postman, State of the API Report, 2025). This falling back to familiar technology prevents them from broadening their view on different more modern technologies. Most existing research focuses on two of the three API styles or emphasizes theoretical discussion rather than hands-on implementation and measurement. Existing theoretical comparisons often fail to account for the nuanced challenges of implementation, such as the learning curve for tooling or specific behaviors like 'over-fetching' encountered in nested data structures. Consequently, there remains a gap in practical, project-based comparisons that assess the trade-offs between performance, developer experience, and maintainability (Saarikoski, 2025).

1.3 Objectives

The objective of this thesis is to design, implement and evaluate a small but realistic web application that allows a comparison of the different API technologies REST, GraphQL and gRPC. The application is inspired by the task management system Trello (see also Chapter 2.1) and serves as a test environment to examine capabilities and limitations of each API under identical conditions.

The application's backend is to include three functionally equivalent API implementations that each expose the same set of features for creating, retrieving, updating and deleting data. This way any observed differences stem solely from the underlying API technology rather than the application logic. This strict isolation of the API architecture as the single independent variable, establishes a controlled environment for experimentation that moves beyond theoretical comparison.

Each API is to be measured and compared across multiple dimensions by conducting benchmarks and assessing aspects such as latency, throughput and payload size. Additionally, quality factors like developer experience, maintainability, and suitability for practical use cases are to be analyzed. In this way, this thesis provides a unique contribution by correlating performance metrics with qualitative "human" costs, such as implementation time and code complexity, for identical features across all three standards.

Based on the results of the above-mentioned comparisons, practical insights and recommendations will be derived. By analyzing the trade-offs between performance, complexity, and usability, the thesis aims to provide clear, evidence-based guidance on why and when to choose either of the three API types in different application contexts.

These objectives enable the thesis to bridge the gap between theoretical discussions and real-world implementation, bringing about a structured and reproducible comparison of three commonly used API technologies.

1.4 Scope and limitations

This thesis covers the development of a small task management application (Trello-like) with the core functionalities being creating, updating, retrieving, and deleting tasks and comments. The

backend is implemented using ASP.NET Core with SQLite, while the frontend is built with Blazor Server to interact with the APIs. The project includes implementing the same functionality using three API styles—REST, GraphQL, and gRPC. This allows a practical comparison of these API technologies.

The focus of this thesis is on comparing the API styles and gaining valuable information for developers unsure which API is best suited for their application. Consequently, the focus is not on the UI design or more advanced features like user management, collaborations on tasks by sharing the task list or other extended functionalities beyond necessary for comparing the APIs. However, the application should be usable on its own as an additional benefit of this thesis.

Certain limitations of this work are that the APIs are only tested with this one type of application. No tests are conducted in larger-scale production environments or with any other types of applications. Therefore, this thesis does not provide insights into API performance under different scenarios or workloads that might occur in these scenarios. Furthermore, the evaluation criteria and their respective weights are subjective and may vary depending on which type of application is developed.

1.5 Research Questions

Based on the problem definition and objectives, the following research questions were formulated to guide the implementation and evaluation process:

1. *How do REST, GraphQL, and gRPC compare in terms of performance efficiency, specifically regarding latency, payload size, and throughput under different load scenarios?*
2. *What are the trade-offs between the three API styles regarding developer experience, implementation complexity, and long-term maintainability and scalability?*
3. *Based on the performance and operational trade-offs, which API architecture is most suitable for specific real-world application contexts?*

These questions facilitate the structure of the benchmarking tests and the qualitative analysis performed in Chapter 5.

1.6 Research Approach

To address the problem definition, achieve the objectives outlined, and answer the research questions defined in the previous chapters, this thesis employs a mixed-methods research approach. This strategy combines constructive research—building a comprehensive knowledge base—with empirical experimentation and qualitative comparative analysis. This combination allows for a holistic evaluation that considers both performance data and the practical reality of software development. The research process is divided into three distinct phases:

1. Constructive Research: The research will include constructing a theoretical basis to analyze the architectural principles and operational mechanism of the three APIs. A Trello-like task management application is developed to serve as a research prototype. To ensure a valid comparison, three functionally equivalent backends were implemented using REST, GraphQL, and gRPC.
2. Quantitative Experimentation: The APIs are benchmarked in a controlled environment using the same tool for all three APIs. Empirical data regarding latency, throughput, and payload size is collected under three specific scenarios to allow for evaluation of performance efficiency.
3. Qualitative Comparative Analysis: The APIs are evaluated based on developer experience, maintainability and scalability, as well as real-word suitability. This analysis relies on gathering concrete performance test result and metrics such as implementation time and Lines of Code (LOC). Furthermore, an assessment of tooling support, type safety, and debugging complexity is conducted.

By integrating quantitative performance metrics with qualitative development insights, this methodological approach ensures that the results are not based solely on theoretical assumptions. Instead, they are derived from verifiable, practical evidence, enabling the formulation of well-founded recommendations for selecting the most suitable API technology in real-world projects.

1.7 Ethical Considerations

This thesis adheres to strict ethical standards and the guidelines for responsible conduct of research. All literature utilized was derived exclusively from publicly accessible sources, and sources have been appropriately cited using the APA 7th edition referencing style.

In compliance with data protection protocols, no sensitive, confidential, or personally identifiable information was processed or entered into Artificial Intelligence prompts.

The AI applications ChatGPT and Google Gemini were utilized to support technical and methodological quality. Their use was strictly limited to technical support by assisting with code debugging and syntax optimization and editorial support by aiding in brainstorming the thesis topic, structure and checking spelling and grammar.

Under no circumstances was any of these tools used to generate research data, analytical results, or the final narrative text. All AI-suggested code was manually verified and tested, and the final written content represents the independent intellectual work of the author or has been appropriately attributed to original sources as either cited paraphrased text or direct quotations.

2 Background of the Project

2.1 Description of the project idea

The idea behind the project is to create an application to be able to compare the three different API technologies: REST, GraphQL and gRPC. For that, an application inspired by the web application Trello is developed.

Waseem (2024) describes Trello as a tool used to manage tasks, projects and team collaboration. It consists of boards for different projects, with cards for individual tasks or ideas sorted into lists that correspond to various project stages. Each card can have additional information like comments, attachments and due dates and can be moved from stage to stage representing progress in the project. Figure 1 shows a screenshot of a basic Trello Board on the Trello Website with three lists.

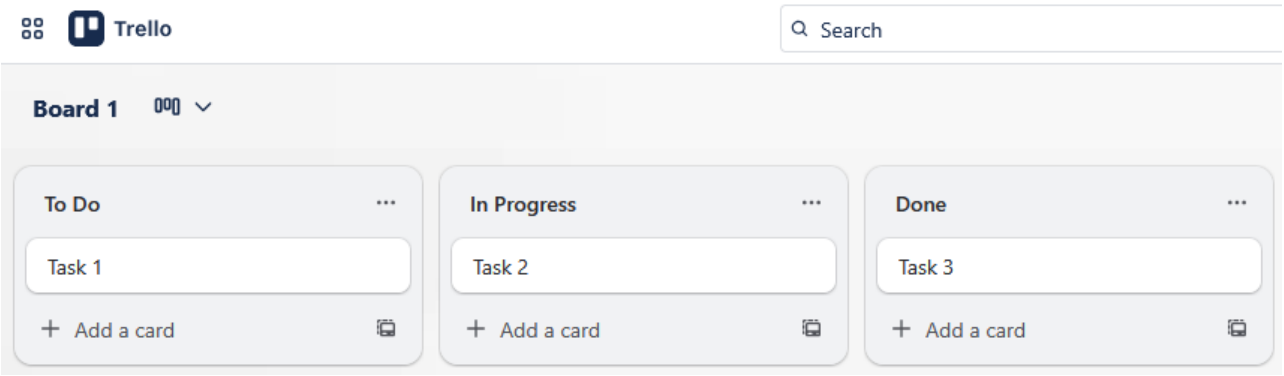

Figure 1. Screenshot of a Trello Board on Trello.com.

This type of application is well suited for testing the APIs because it involves the operations create, read, update, and delete (CRUD) as well as nested queries with the possibility to attach several comments to a single task. Figure 2 shows the details page for an individual task. Each task can have a creation date and a due date, a detailed description, as well as comments.

Another reason this application fits the requirements is because it is small enough to be implemented in limited time but also complex enough to stress APIs in order to test and compare them.

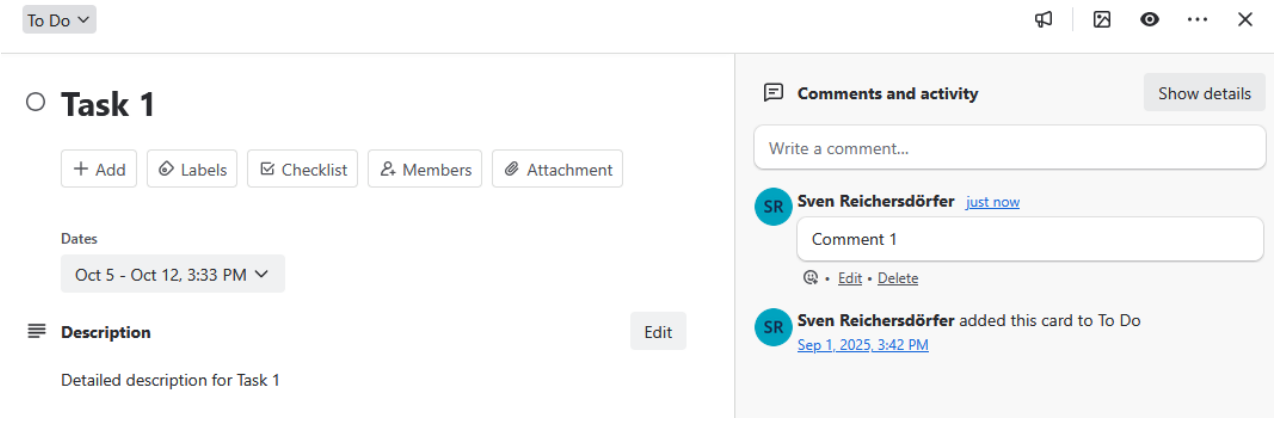

Figure 2. Screenshot of Trello Task Details on Trello.com.

Although the performance measurements are carried out independently of the frontend using automated tests, the frontend is included to verify the functional equivalence and practical usability of all three APIs. It demonstrates how REST, GraphQL, and gRPC can be consumed within the same application context and ensures that performance comparisons are based on identical functionality. The frontend thus serves as both a validation tool and a practical showcase of API integration.

2.2 Technology overview

The following chapters provide a brief overview of the technologies used in this project.

2.2.1 ASP.NET Core

ASP.NET Core is a modern and open-source framework for creating web applications and APIs across platforms. It was built by Microsoft mainly for speed and flexibility and it is a redesigned evolution of ASP.NET that works seamlessly on different platforms like Linux, macOS, and Windows (ScholarHat, 2025).

2.2.2 EF Core

As described by Patel (2024), Entity Framework Core (EF Core) is an Object-Relational Mapping (ORM) framework from Microsoft which enables .NET developers to use C# objects to work with databases instead of using raw SQL. It simplifies data access with features like cross-platform support, LINQ integration, change tracking, and schema migrations, making database management easier and more efficient for application development. With EF Core a code first approach is used,

meaning the classes for the data are created first and then migrated to the database. This way, when changes in the code are made, they are easily synced with the database (Maze, 2024).

2.2.3 Blazor Server

Blazor is an open-source web framework supported by Microsoft that facilitates the creation of interactive web UIs in C# rather than JavaScript. The name stems from the combination of "Browser" and "Razor" which is the .NET HTML view rendering engine (Hadzima, 2023).

According to Meghna (2024), there are three distinct categories and corresponding Visual Studio templates when it comes to Blazor: Blazor Server, Blazor Web Assembly and Blazor Web App. For this application, a Blazor Server is used. Blazor Server is a hosting model where the application logic runs entirely on the server, with the UI updates sent to the browser via a real-time connection. Compared to Blazor Web Assembly, it offers faster initial load times and does not require downloading the entire application to the client. The server-side execution allows direct access to the database via EF Core, and the focus stays on the API comparison not on browser limitation.

Blazor Server uses a standard ASP.NET Core application with which functionality on the side of the server can be integrated. SignalR is used to communicate constantly between the Blazor Server application and the browser. The pages on the client side can be created using Razor components or Razor pages (Beres, 2025).

2.2.4 Razor Pages and Components

Razor Pages is a page-oriented programming model in ASP.NET Core. Each Razor page comprises of a `.cshtml` file and a `.cshtml.cs` file with a specific route. Each page is backed by a `PageModel` class that defines handler methods such as `OnGet()` and `OnPost()`, which means Razor Pages primarily respond to GET and POST requests (Microsoft, Razor Pages architecture and concepts in ASP.NET Core, 2025).

Razor Components form the foundation of Blazor Server and combine C# and HTML within .razor files to create interactive, reusable UI elements. A component becomes a routable page when decorated with an `@page` directive, enabling a hybrid, component-based UI architecture (Microsoft, ASP.NET Core Razor components, 2024).

In this project, Blazor Server uses Razor Components to render the user interface, while all data operations are executed through backend APIs implemented with REST, GraphQL, and gRPC. Although Razor Pages traditionally support only `GET` and `POST`, this limitation does not apply here, as all CRUD operations are handled by these APIs rather than through page handlers.

2.2.5 MudBlazor

Attique (2023) and Kumar (2024) describe MudBlazor as a UI and component library for the Microsoft Blazor framework used for frontend development. It allows the developer to build modern and interactive web applications inside the web development framework Blazor, by providing a rich component set, customization possibilities, and responsive design. The component set includes a variety of pre-built components, among others, buttons, input fields, dialogs, menus, and tables. MudBlazor can be added to the C# project by simply installing the NuGet package in the Package Manager of Visual Studio.

2.2.6 SQLite

SQLite allows for fast data access and has the advantage that the database is stored as a single disk file. This way, data can be queried without having to load the complete set of data into the memory (Kanjilal, 2024). Furthermore, the database file can easily be copied for backup or migration purposes (Saleem, 2024).

Since EF Core comes with a dedicated provider for SQLite, no additional complex configurations are necessary to connect to a SQLite database but only the NuGet package has to be installed and configured in the project (Microsoft, Entity Framework Core Getting Started with EF Core, 2023; Microsoft, Entity Framework Core SQLite EF Core Database Provider, 2024).

2.3 API styles and their relevance

In the implemented task management system, the API layer forms the core channel for the communication between the Blazor Server frontend and the ASP.NET Core backend. All operations related to tasks and comments—such as creating, updating, retrieving, and deleting—are exposed via APIs. A fair and practical comparison is enabled by implementing the same functionality for all three different API styles: REST, GraphQL, and gRPC. Each of these API styles manages data exchange and interaction in diverse ways, bringing their own benefits and challenges to this project.

In this project, REST serves as the natural baseline for comparison. As the widely accepted industry standard for web APIs it offers a straightforward way to perform CRUD operations on resources such as tasks and comments (Mikula, 2023). This baseline enables an accurate comparison and measurement of the relative deviation of the other APIs in performance, developer experience, maintainability and scalability, as well as real-word suitability.

GraphQL is included to directly address some of REST's structural limitations regarding related data. Since the implemented task management system relies on a one-to-many relationship where tasks can contain multiple comments, GraphQL's capability to fetch nested data in a single query can directly solve some of REST's issues (Stemmler, What is GraphQL? GraphQL introduction, 2021).

gRPC, in turn, is selected for its high performance and binary message format, which can offer benefits in scalability and responsiveness, if the system were to be extended into a microservice architecture, that REST and GraphQL might not be able to offer to the same degree. It represents the shift towards high-performance binary communication protocols (Postman Team, What is gRPC?, 2023).

The selection of these three technologies also reflects a part of the historical evolution of web API architectures, moving from the mature industry standard (REST) to the flexible data-querying solution (GraphQL) and finally to the modern, high-performance RPC framework (gRPC) (Motunrayo, 2024).

Comparing these three API styles within the same implementation allows an assessment of how different communication approaches affect performance, data handling, and development complexity in a realistic web application scenario. The evaluation focuses on how efficiently each style handles operations typical for the application, such as handling nested task–comment relationships, and the frequent data updates between client and server typical in a modern web application.

3 Theoretical Basics

3.1 The Role and Evolution of Application Programming Interfaces (APIs)

3.1.1 Concept of APIs in Software Engineering

Team (2023) states that “APIs serve as a bridge for software developers to interact with external software components or resources”, highlighting their role in enabling seamless communication and data exchange between different systems. An application programming interface (API) is a set of protocols and rules, essentially like a common language, that facilitates applications to communicate or exchange data with each other. APIs can allow developers to access information from platforms or applications efficiently, without the need to develop components from the ground up or understanding details about how the specific application is set up internally (Postman, What is an API?, n.d.; Mulesoft, n.d.). This, in turn, classifies APIs as an example of data abstraction since they allow communication amongst different systems by only exposing necessary data and functions. This abstraction enables developers to more efficiently build complex systems and simplifies the integration process (CelerData, 2024). Jacobson, Brail and Woods (2011) describe an API as a contract, because once in place, developers are inclined to use the API since they know they can rely on it, which, as a consequence, increases the use of the API and makes the communication between provider and consumer more efficient. The “contract” ensures documented, consistent, and predictable interface between them.

As noted by Goodwin (2024), APIs are used in most modern digital services, such as mobile apps and cloud platforms. For example, travel websites use APIs to request real-time availability from airlines and display it to users. Another example is universal log in, which enables users to authenticate via Google or Amazon accounts on other websites.

In general, APIs take different roles in modern enterprise architectures. They enable communication and integration and promote modularity and microservices, by serving as mechanism to communicate and exchange data, and reducing dependencies by using clear contracts. Furthermore, APIs ensure standardization by providing clear documentation and adhering to industry standards. Security and access control, which is provided by APIs, allow for the integration of authentication

and authorization mechanism. By leveraging existing functionality and data, flexibility and scalability is ensured and reusable components and standardized interfaces optimize development and operations (Manager, 2025).

In today's industry, organizations are increasingly embracing an "API-first" mindset, with 82 % of companies having adopted some level of API-first development and 25 % being fully API-first. This means that APIs are seen as core products rather than add-ons which is reflected by 65 % of organizations reporting, that they generate revenue from their API programs (Postman, State of the API Report, 2025). Meanwhile, the global API management market is expected to make it to USD 7.1 billion in 2025 and expand at a compound annual growth rate (CAGR) of roughly 26.5 % through 2034 — meaning the market is expected to expand by roughly 26.5 % each year when averaged over the full period (Kharrati, 2022). Together, these facts highlight that APIs are not just technical enablers but a foundation to modern application architectures and business models.

3.1.2 The Client-Server Model and Basic API Workflow

Nath (2023) explains, that APIs typically work with communication between a client and a server in a request and response style. This typical request-response model consists of a client that sends a request to a server which responds with a corresponding response. However, as Sharma (2023) maintains, several steps are performed between request and response. After the client (for example a web browser or an application) sends the HTTP request to a URL or endpoint, the request has to be parsed and processed by the server. This step may include operations like retrieving data from a database, manipulating data or validating the request. Once this step is completed, the server generates a response which may, amongst other things, include the requested data or result of a potentially executed operation. The server sends the response back to the client, which processes it accordingly after parsing and uses the information in the response to for example update the user interface or render an HTML page. From this point on, the pattern is repeated as necessary for the application.

Based on the previously described request and response pattern, an API can be divided into four basic components depicted in Figure 3.

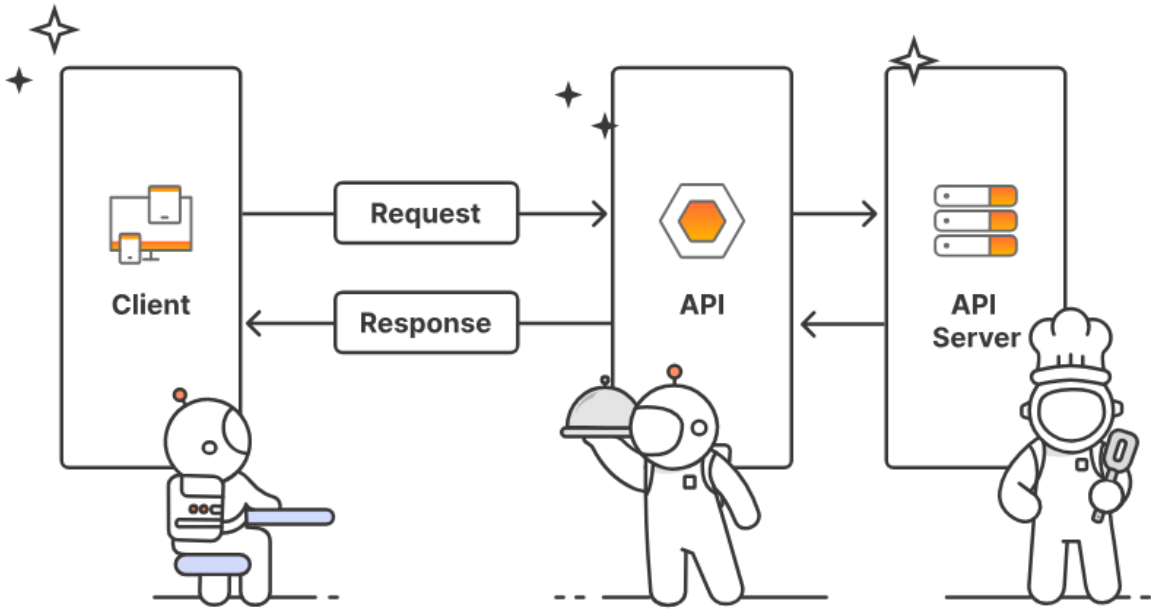

Figure 3. General schema of an API (Postman, What is an API?, n.d.).

The API client initiates the communication by sending a request which can for example be triggered by a button pressed by a user or external sources such as other applications. This client could be a mobile application, web browser or any other device capable of network requests (Nath, 2023).

The API request typically consists of several components, most importantly the endpoint that provides access to the information. Fateh (2025) describes the API endpoint, which is mostly provided as a URL, not as the resource itself, but rather a specific location in the API's server architecture where client and server can interact. The method or type of operation the client would like to perform is also comprised within the API request. Additionally, a request includes parameters with specific instructions for the API. This additional information is made up of headers, which contain additional metadata about the request such as the content type or authorization tokens. Moreover, an optional body or payload contains data or content for the server to be processed (Postman, What is an API?, n.d.; Sharma, 2023).

The API server exposes its endpoints to receive requests from a client. Once the request is received, it is handled and processed. After the request has been processed, an appropriate response is generated and returned to the client (Innocent, What is an API Server?, 2025). According to Nath (2023), the typical content of a response includes a three-digit status code which signifies the outcome of the request. Typically, as mentioned by Fateh (What is an API call?, 2025), success codes are indicated by “2xx”, with 200 being the code for a successful request. Error codes typically start with “4xx” and status codes for server errors with “5xx”. Nath (2023) continues to explain that equivalent to the request, the response contains headers with additional metadata and, again, an optional body with data or content sent from the server. This could for example be HTML content, JSON data, or binary files which was requested by the client.

3.1.3 History of Web API Architectures

As explained by Mikula (2023), the first mention of an outline of what could be described as an API was mentioned in the book “The Preparation of Programs for an Electronic Digital Computer” published in 1951 by Wilkes and Wheeler. The concept of APIs evolved in the 1960s and 1970s with the growing popularity of computers and the demand for multiple software systems to communicate with each other (Miller, 2023). In a paper called “Data structures and techniques for remote computer graphics“ by Ira W. Cotton and Frank S. Greatorex Jr., the term application program interface first shows up but at that point, APIs remain as local interfaces on the same computer. In his 1989 paper “Information Management: A Proposal”, Tim Berners-Lee proposed a standardized interface between a web browser and a webserver (HTTP) and later, the success of the web in the year 2000 led to ideas emerging about how to use the web beyond just for web browsers (Wilde, 2025).

In 2000, when the company Salesforce released what many consider to be the first modern API, the computer scientist Roy Fielding established Representational State Transfer (REST) as a concept in his doctoral dissertation. Its emphasis on a stateless resource-oriented architecture allowed for standardized communication between devices across the internet (Mikula, 2023). A couple of years later, other protocols and standards arose to address specific requirements and use cases. Companies such as Flickr, Facebook and Twitter published APIs to for example access or share user information. Soon after, Amazon and other organizations started moving more towards

cloud-based solutions and establishing APIs as core business strategies with Amazon's Web Services released in 2007 (Hawkins, 2020).

Prior to 2008, APIs were generally regarded as a background component that support larger systems. That perspective shifted when Twilio pioneered the "API-as-a-Product" model, making the API its core business (e.g., communication services). This shift created the modern API marketplace, with companies like Stripe and Algolia following suit. The concurrent mobile revolution cemented the API's role. Mobile devices drove massive growth in data creation and sharing (photos, videos), making APIs essential for powering modern mobile applications and connecting virtually every object to the internet (Lane, 2019; Mikula, 2023). As applications became more data-hungry and demands for greater efficiency increased, new approaches to API design became necessary. In 2012, this need for a more effective data-fetching method lead Facebook to create GraphQL which was later published in 2015 after a period of internal use (Stemmler, What is GraphQL? GraphQL introduction, 2021).

The foundational concept of Remote Procedure Call (RPC), which allows a program to execute code on another machine as if it were local, is a decades-old idea dating back to the late 1970s and early 1980s (AWS, What's the Difference Between RPC and REST?, n.d.). This paradigm was used by Google, first developing a high-scale, internal RPC system called Stubby around 2001 to connect its massive network of internal services. In 2015, Google released the next generation of this technology and called it gRPC (Postman Team, What is gRPC?, 2023).

3.2 Representational State Transfer (REST) Fundamentals

3.2.1 Core Architectural Principles of REST

Representational State Transfer, also known as REST, is the most used architecture for web APIs (Mulesoft, n.d.). REST APIs are generally considered straightforward to implement, allowing programmers to use almost any programming language and data format. However, alignment with the six core REST design principles, also known as architectural constraints, is mandatory. These constraints ensure clean, well-documented and standardized code as well as optimal reliability, scalability, and extensibility (IBM, 2025).

Kiprono (2023) describes the first architectural constraint as uniformity of the interface. This means that REST requires uniquely identified resources, consistent representations for modifying them and self-descriptive messages describing how to process and interact with those resources.

Secondly, the server and client have to be separated so that each side can be implemented independently of each other and changes to one side have no effect on the other side (Bennett, 2024).

REST APIs are also considered stateless, which means that all the information needed for processing a request is included in the request itself and no additional data is required to be stored on the server side (IBM, 2025).

As Bennett (2024) explains, caching is another architectural constraint of REST. The API must specify whether and how long a response can be cached and a server response must contain information whether or not caching is allowed for the delivered resource.

The requests and responses in REST APIs go through different layers, however, each component cannot see beyond the layer they are immediately interacting with. This implies that the REST API needs to be designed in a way that neither server nor client can distinguish whether they communicate with the end application or an intermediary (Kiprono, 2023).

Lastly, an optional constraint is code on demand, meaning that an API can, instead of sending a response containing static resources, return executable code, for example in the shape of Java applets (IBM, 2025).

3.2.2 Resources, URIs, and HTTP Methods

In principle REST APIs function the same way as browsing the internet, meaning that clients contact the server when a resource is required. To perform CRUD operations, REST APIs use standard HTTP verbs like `GET`, `POST`, `PUT`, `DELETE` (AWS, What is a RESTful API?, n.d.). Figure 4 shows a model of REST API with the Client and Server as well as the Request and Response.

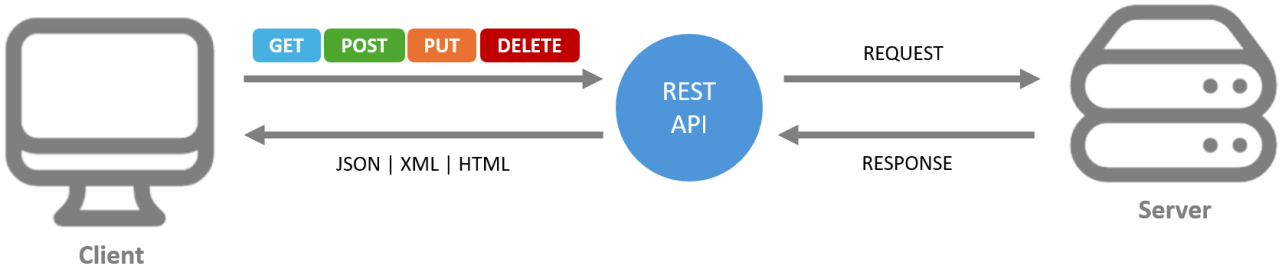

Figure 4. Model of a REST API with Client and Server.

The different HTTP verbs each serve a distinctive purpose. The `POST` verb creates a resource, `GET` reads or fetches a resource, `PUT` allows to update a resource, and `DELETE`, as implied by the method name, deletes a resource (Salma Alam-Naylor & David, 2024).

Besides the HTTP method, a client request contains so-called resource endpoints, each being identified by a Uniform Resource Locator (URL) which clearly states the location of the resource. Also, headers and parameters play an important role in requests since they include metadata, authorizations, caching information, and so forth (IBM, 2025).

The response of the server contains a three-digit status code (see also Chapter 3.1.2) as well as a body, in case data was requested, for example with the `GET` method. The requested data returned by the server can be supplied to the client in a variety of formats, including HTML, XML, or JSON. For modern web services, JSON is the most commonly used format (geeksforgeeks, 2025).

Figure 5 shows an example request containing the GET method to query data about a user with the number 42 from the server. Figure 6 shows a possible response from the server, containing a status code, the content type as well as the requested user data in JSON.

```
GET /api/users/42 HTTP/1.1
Host: example.com
Accept: application/json
```

Figure 5. Example of a GET request with REST.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 42,
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

Figure 6. Example of a response from a GET request with REST.

3.2.3 Data Transfer and Common Issues

The utilization of standard JSON and HTML makes REST APIs comparably easy to implement and allows programmers to use virtually any programming language as well as a variety of data formats (Center, What is a REST API? A Comprehensive Guide , 2025). The previously described architectural constraint of server and client separation also ensures efficient scaling and the statelessness can help to reduce server load (AWS, What is a RESTful API?, n.d.).

Panchal (2024) and Pawar (2025) state that, however, APIs also come with certain drawbacks. Complex queries and multiple endpoints can result in increased project complexity and more challenges in managing the project. Additionally, over- or under-fetching are the predominant drawbacks of REST.

Over-fetching occurs when the API response contains more data than required by the client. This can lead to higher, unnecessary bandwidth and data usage and can negatively impact REST's performance. Under-fetching, on the other hand, means that the client requires more data than what a single endpoint is able to provide, forcing the client to make multiple requests, leading to increased latency and network overhead (Okpala, 2024).

3.3 GraphQL Fundamentals

3.3.1 GraphQL as a Query Language, Not an Architecture

Dilgemani (2025) describes that in contrast to REST architecture, GraphQL is not an API specification but a query language utilizing a user defined type system for the data. By providing a intuitive and flexible syntax and system for describing data interactions and requirements, it can be used to build client applications (GraphQL, GraphQL Spec Overview, 2025). While a REST query will extract all the data from an endpoint, GraphQL only receives what is requested in the query. Consequently, GraphQL allows clients to query a unified endpoint and ensure they receive only the specific data required and avoid over- or under-fetching (Hygraph, What is GraphQL?, n.d.). This makes GraphQL a more product centric language and gives clients more control over their data requirements (Miquissene, 2020).

3.3.2 The GraphQL Type System (Schema)

GraphQL works with a type system, describing what data can be queried from the API. The schema is a collection of those capabilities and can be used by clients to send queries and receive predictable data in return (GraphQL, Learn / Schemas and Types, n.d.). The schema acts like a contract between server and client and defines how clients can request or change data or what the API can or cannot do (Apollo, Tutorials/Lift-Off-Part-1/Schema definition language (SDL), n.d.).

As described by (Hygraph, Learn / Schemas and Types, n.d.), the schema is developed in a language called Schema Definition Language, defined by GraphQL itself. It mainly consists of `types` and their `fields` and supports typical, build in scalar types such as Int, Float, String, Boolean and ID. Furthermore, non-nullable fields and lists can be used as a types for fields (Burk, 2017).

3.3.3 Queries, Mutations, and Resolvers

GraphQL makes the important differentiation between the two mechanisms: query and mutation. The type of operation is indicated in the beginning of each GraphQL operation string. Queries can be compared to the HTTP verb `GET` since they are used to fetch data.

The following Figure 7 and Figure 8 show an example query with a possible response. The query asks for the field `user` and for the fields `name` and `email` of the returned `user`. The data types are predefined and, on request, validated and executed against the schema (Stemmler, GraphQL Mutation vs Query – When to use a GraphQL Mutation, 2021).

```
query {
    user (id: 1) {
        name
        email
    }
}
```

Figure 7. Simple GraphQL Query.

```
"data": {
    "user": {
        "name": "John Doe",
        "email": "johndoe@mail.com"
    }
}
```

Figure 8. GraphQL response to a simple Query.

Mutations, on the other hand, closely represent the HTTP verbs `POST`, `PUT`, and `DELETE` and are used to modify server-side data. The desired response data is stated inside the mutation so, as can be seen in Figure 9, `id`, `name`, and `email` are contained inside the mutation thus returned in the response seen in Figure 10.

```
mutation CreateUser($userInput: User!) {
    createUser(userInput: $userInput) {
        id
        name
        email
    }
}
```

Figure 9. Simple GraphQL Mutation.

```
"data": {
    "createUser": {
        "id": "1",
        "name": "John Doe",
        "email": "johndoe@mail.com"
    }
}
```

Figure 10. Response to GraphQL Mutation.

In order to pass input values into queries and mutations, GraphQL supports variables which allow for the use of the same mutation or query for different tasks. In Figure 9, the variable used to input user data is called `userInput`. The definition and possible content of the variable can be seen in Figure 11.

```
{
    "userInput": {
      "name": "John Doe",
      "email": "johndoe@hygraph.com"
    }
}
```

Figure 11. Definition of `userInput` GraphQL variable.

Variables ensure a decoupling of mutation or query code from the source of the inputs as well as improve readability and type safety (Kramer, GraphQL Queries & Mutations: A Guide, 2024).

Another important component of GraphQL are Resolvers. They act as the link between a client's query and the underlying data sources, determining how each field's data is fetched and formatted. Resolvers are defined within the GraphQL schema and handle returning different types of data ranging from simple scalar values to complex objects, which are retrieved from for example databases, APIs, or external services. Overall, they ensure that every field in a query is resolved into the exact structure expected by the client (Bhayana, 2025).

3.3.4 Operational Considerations

GraphQL is typically used in environments with rapidly changing frontend requirements and more complexity. Furthermore, it is useful for applications that may operate with slow or unreliable internet connections due to the reduced communication between server and client (Mulesoft, n.d.; Postman, What is an API?, n.d.).

While GraphQL has its benefit in scenarios where flexibility and efficiency are needed, the necessity to parse and execute queries can cause more overhead. Additionally, since GraphQL requires a different approach to API usage and design, there exists a learning curve for REST developers (Pawar, 2025). Another downside of GraphQL is its more challenging approach to traditional HTTP caching, especially with highly dynamic queries (Giroux, 2019). GraphQL queries and mutations are often sent as `POST` requests to a single endpoint so, since the single endpoint returns different responses, the response cannot be cached with the URL as the identifier, as it is typically done in HTTP. While there are different solutions to solve this problem, caching remains more complicated when using GraphQL (Losoviz, 2021).

3.4 gRPC Fundamentals

3.4.1 Remote Procedure Call (RPC) Concepts

RPC stands for Remote Procedure Call and is a software communication protocol used by programs located on different computers or networks to request services from each other (Gillis, 2024). It does that in a way that it behaves as if the service (procedure) were located locally. By

abstracting the network complexity, developers are able to focus on application logic without being distracted by the underlying details of remote procedure calls (Studio, 2024). RPC functions in a client-server model thus consists of client, server, a communication protocol used for exchanging messages as well as a stub (client proxy) and skeleton (server proxy). The stub represents the remote service locally for the client to interact with and the skeleton is a representation of the server's interface to the client (Chandima, 2019).

A specific implementation of the RPC framework is gRPC which was built by Google. It was developed to connect distributed applications in a fast and high-performing manner. To meet this requirement, gRPC was developed. Its data serialization offers low latency and high throughput and the use of the HTTP/2 protocol provides features like code generation as well as bidirectional streaming capabilities (Nosowitz, What is gRPC?, 2024). These qualities make gRPC best suited for environments requiring real-time streaming capabilities or high performance, as well as applications like microservice architectures, where low latency is crucial (Goodwin, 2024).

Synytsia (2021) points out that one drawback of gRPC is its lack of human readability, stemming from the fact that, rather than more transparent formats like XML or JSON, it uses a binary format for data transmission. Moreover, when compared to REST and GraphQL, working with protocol buffers can require more time to get acquainted with and finding tools for dealing with HTTP/2 can be more challenging.

3.4.2 Protocol Buffers (Protobuf) and Data Serialization

Protocol Buffers (Protobuf) are used as the interface definition language and encode data in an efficient binary format. Data serialized into this format is significantly more compact than equivalent data in text-based formats like JSON or XML (Bello, 2024). This leaner form of serialization enables faster transmission and improved performance (Center, gRPC: A Comprehensive Guide to Modern API Development, 2024). The structure of the data is defined by the programmer in a specific number-based schema in a `.proto` file. The `.proto` file consists of messages, defining the structured data of inputs and outputs as well as services defining the API methods. Figure 12 shows an example of a message in a `.proto` file defining required and optional fields with their respective data types and numerical identifiers. An example of a

definition of a service to retrieve a customer, as defined in the message in Figure 12, can be seen in Figure 13.

```
message Customer {
    required int32 id = 1;
    required string name = 2;
    optional string address = 4;
}
```

Figure 12. Example of a message in a `.proto` file.

```
service CustomerService{
    rpc GetCustomer (GetCustomerRequest)
        returns (Customer);
}
```

Figure 13. Example of a service in a `.proto` file.

Once the structure is defined in the `.proto` file, a suitable Protobuf compiler like Protoc can generate code for a variety of different programming languages (Bello, 2024). This automatically generated code allows structured data to be read and written in a seamless and efficient way across platforms (Saraiya, 2024). To ensure type safety and consistency, the same `.proto` file is used for the backend and the frontend and acts as a contract between the two (Hegde, 2024).

3.4.3 HTTP/2 and Transport Layer Efficiency

According to Bhayani (n.d.), when Google developed gRPC they deliberately built it on HTTP/2 rather than the more widely adopted HTTP/1.1. This decision could be explained by several of the advantages HTTP/2 has over HTTP/1.1. Firstly, HTTP/2 uses more compact binary encoded requests compared to the text-based requests of HTTP/1.1. Tech School (2020) adds that the binary decoding is a good combination with the previously described and also binary encoded Protobuf. As Jadhav (2024) states, this enables a more efficient network data transfer, supporting gRPC's performance requirements.

Secondly, HTTP/2 supports multiplexing, allowing a single TCP connection to carry multiple requests and responses simultaneously by interleaving them across independent streams. The functionalities like server, client or bidirectional streaming, described in the following Chapter 3.4.4, rely on the fact that gRPC utilizes HTTP/2 which supports streams (Bhayani, n.d.).
More compressed request and response header metadata reduces encoding time and header size, in turn further improving gRPC's performance (Jadhav, gRPC deep dive : Efficient network communication using HTTP/2, 2024).

3.4.4 Communication Methods (Unary and Streaming)

As mentioned in the previous chapter, gRPC uses the streaming capabilities provided by the HTTP/2 protocol. HTTP/2s streaming is extended by gRPC to provide streaming RPC calls (Jadhav, gRPC deep dive : Introduction to gRPC, 2024). Streaming allows to send and receive multiple messages in a single connection (Innocent, How gRPC Streaming Can Make Your APIs Faster and More Reliable, 2025).

As described by Farokhi (2022) and Innocent (How gRPC Streaming Can Make Your APIs Faster and More Reliable, 2025), gRPC offers four different types of streaming. Unary streaming is similar to the regular HTTP request and response pattern. A client sends one request to the server and the server returns one response to the client. In server streaming the client sends only one request to the server and receives a streaming response from the server. The server can send responses continuously as they are ready, without waiting for the client to request each one. Client streaming is the opposite, where the client initiates the remote procedure call and then sends streaming messages or requests to the server without waiting for acknowledgement from the server. Lastly, in bidirectional streaming, messages can be sent in both directions by the client and the server in parallel. There is no strict order and the messages can be sent and received independently of each other.

3.5 Criteria for evaluation

3.5.1 Criteria description

The three APIs are evaluated based on four different criteria. Each criterium is assigned an individual weighting which produces a quantifiable score and enables a clearer evaluation of the results and more precise comparison between the APIs. The evaluation criteria are listed and explained in the following chapters and the bracket behind each criterion stats the respective weighting percentage.

3.5.2 Performance (40%)

Performance focuses on how efficiently each API handles requests under different workloads. Table 1 shows the sub-criteria for the APIs performance analysis with the respective sub-criteria weight and explanation.

Table 1. Performance sub-criteria with weight and explanation.

Performance (40%)	**Sub-criterion**	**Weight**	**Sub-criterion Explanation**
	Latency	20%	Measures response time, especially the impact of small vs. large queries and concurrent load.
	Payload Size	10%	Examines the bytes transferred in each request and differences due to over-fetching or under-fetching data.
	Throughput & Resources	10%	Assesses requests per second handled and the necessary CPU and memory usage under load.

3.5.3 Developer Experience (30%)

Developer experience considers how pleasant and straightforward the use of the API is from a developer's perspective. The category further divides into the sub-criteria listed and explained in Table 2.

Table 2. Developer Experience sub-criteria with weight and explanation.

Developer Experience (30%)	**Sub-criterion**	**Weight**	**Sub-criterion Explanation**
	Ease of Implementation & Implementation Time	10%	Measures the time/effort required to implement a basic application, complexity of adding features, and lines of code.
	Learning Curve & Tooling Support	10%	Evaluates the availability and usability of tools, documentation quality, and onboarding difficulty.
	Flexibility for Clients	10%	Focuses on features like type safety, schema enforcement, and the client's ability to shape responses.

3.5.4 Maintainability & Scalability (20%)

Maintainability and scalability examine how stable the API is under operation and how well it supports long-term growth. Table 3 shows the sub-criteria for maintainability and scalability with weight and explanation.

Table 3. Maintainability & Scalability sub-criteria with weight and explanation.

Maintainability (20%)	**Sub-criterion**	**Weight**	**Sub-criterion Explanation**
	Versioning & Evolution	10%	Compares approaches like REST endpoint versioning, GraphQL schema evolution, and gRPC contract updates.
	Debugging & Error Handling	10%	Assesses logging complexity and the clarity of error reporting.

3.5.5 Suitability (10%)

Real-world suitability evaluates the practical applicability of the API in different environments. The explanation of the only sub-criterion of this category can be found in Table 4.

Table 4. Suitability sub-criteria with weight and explanation.

Suitability (10%)	**Sub-criterion**	**Weight**	**Sub-criterion Explanation**
	Use-Case Fit	10%	Determines which API style is better for different scenarios and applications and considers practical trade-offs in production.

4 Practical Implementation of the Project

4.1 System architecture and design

The project is divided into the frontend, which is the part that can be seen by the user of the application, the backend which contains all the necessary API function and a library to share data and functions used by both the frontend and the backend. The frontend consumes all three APIs to connect to the backend. Figure 14 shows the architecture of the system for this project.

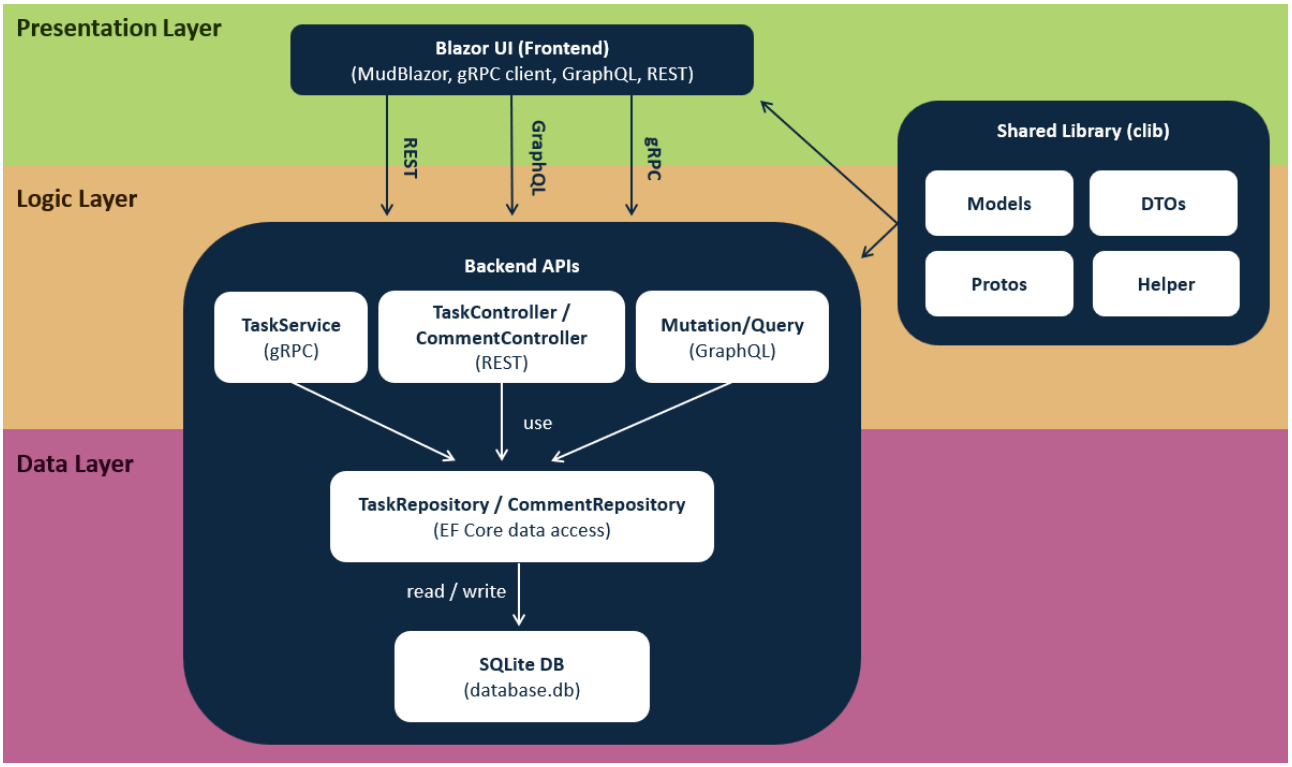

Figure 14. Schema of the system architecture in this project.

As illustrated in the figure, the architecture utilizes a shared library (`clib`), positioned between presentation and logic layer, which enforces consistency across the stack. However, the data exchange between frontend and backend is solely done by utilizing one of the three APIs.

Furthermore, the Data Layer implements a unified Repository Pattern on which all three API backends rely. This isolates the API transport layer as the only variable and ensures that potential performance differences are attributed exclusively to the API architecture itself rather than variations in data access logic.

4.2 Database schema

Figure 15 shows the database schema with the relation between TaskItems and Comments as well as the datatype of each property in the respective class.

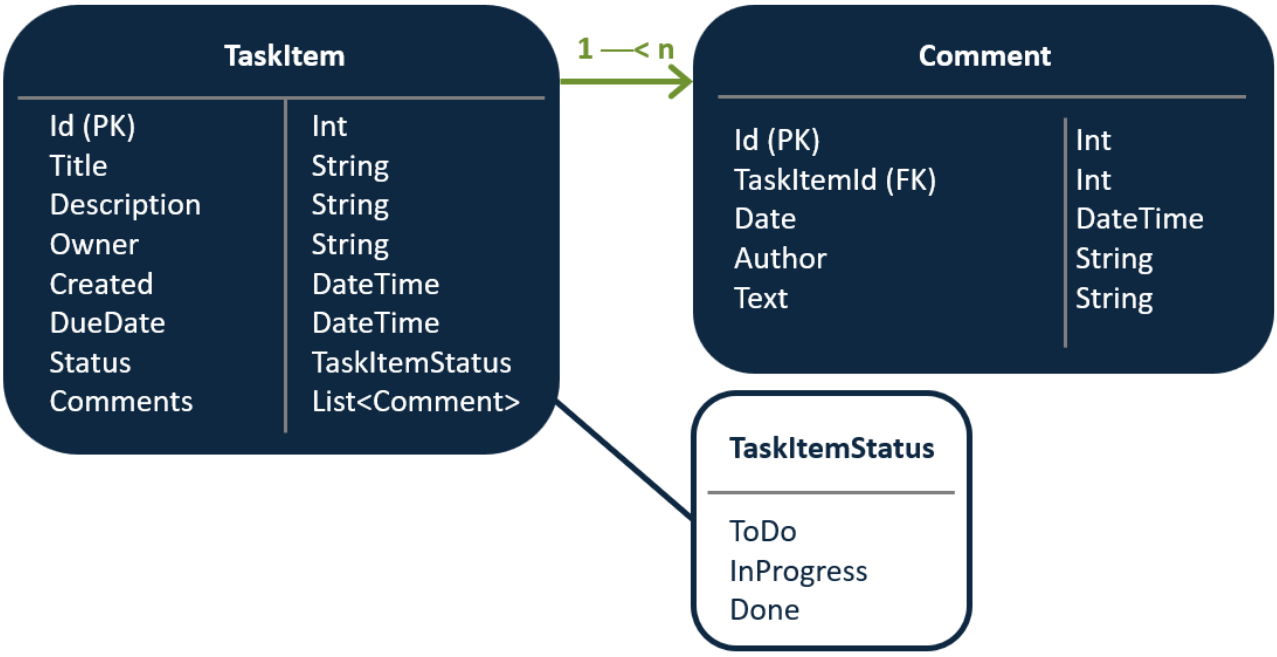

Figure 15. Schema of the Database containing tasks and comments.

The database schema follows a simple relational design consisting of two entities: `TaskItem` and Comment. A one-to-many (1:n) relationship exists between `TaskItem` and Comment, where each task can have multiple comments, but each comment belongs to exactly one task. This is implemented using the `TaskItemId` foreign key (FK) in the `Comment` table. Each `TaskItem` and `Comment` is identified by a unique primary key (PK). This primary key, which is conventionally called Id in the context of SQLite, is automatically generated by EF Core once a new entity is saved to the database (Microsoft, Entity Framework Core - Keys, 2022).

4.3 Backend implementation

The following chapters describe structure and setup of the backend and the implementation of the individual APIs in more detail.

4.3.1 Backend structure

The backend is implemented in a single solution in an ASP.NET Core Web App in Visual Studio. This solution includes all three APIs as well as the SQLite database. A class library is used to provide the necessary data structures and models for both the Blazor server (frontend) and the .NET Core Web Application (backend).

The .NET Core Web Application allows a comparably easy integration of the different APIs and the SQLite database. In `Program.cs`, a builder is defined that allows the database context and various services — such as the REST controllers, the GraphQL server, or the gRPC service — to be added to the application.

In the `launchSettings.json` file, the server URL is defined. This project uses the localhost as a server and no public URL which simplifies the testing of the APIs as the main purpose of this project.

4.3.2 REST endpoints

The REST endpoints are created in the `TaskItemController.cs` and `CommentController.cs`. Each HTTP method `GET`, `POST`, `PUT` and `DELETE` has at least one corresponding method in the controller class. Figure 16 shows the method in the task item controller with which all tasks can be requested by calling a `GET` method with the correct URL and end point `TaskItem`.

```
[HttpGet] // GET: api/TaskItem
public async Task<ActionResult<IEnumerable<TaskItem>>> GetTasks()
{
    _logger.Trace("HttpGet: GetTasks");
    // Return all tasks
    return await _repo.GetAllTasksAsync();
}
```

Figure 16. `GetTasks` method in `TaskItemController` class.

With the respective method attribute (see square bracket before method definition in Figure 16), the method gets declared as a HTTP method and can be used as an endpoint for REST communication. By additionally providing the id of the `TaskItem` as a parameter in the method attribute, a

single task can be queried via REST, see Figure 17. In the same way, the other required HTTP methods to update, create or delete `TaskItems` and `Comments` are created.

```
[HttpGet("{id}")] // GET: api/TaskItem/{id}
public async Task<ActionResult<TaskItem>> GetTask(int id)
{
    _logger.Trace($"HttpGet: GetTask with ID: {id}");
    // Retrieve task by ID
    var task = await _repo.GetTaskByIdAsync(id);
    if (task == null) return NotFound();
    return task;
}
```

Figure 17. `GetTask` method in `TaskItemController` class in the backend.

The attributes `[ApiController]` and `[Route("[controller]")]` in the beginning of both controller classes ensure that the controllers behave like a proper REST API controller and define the route pattern. The route tells ASP .NET Core how to map HTTP requests to this controller (Microsoft, Create web APIs with ASP.NET Core, 2024).

4.3.3 GraphQL schema

As demonstrated in the GraphQL Fundamentals chapter, the differentiation between queries and mutations is important when it comes to GraphQL. That is why in the GraphQL backend, two separate classes are created to handle queries and mutations. Figure 18 shows the mutation to add a new task to the `TaskRepository` and consequently the SQLite database.

```
/// <summary>
/// Adds a new task based on GraphQL input.
/// </summary>
public async Task<TaskItem> AddTask(GraphQLAddTaskInput input,
    [Service] TaskRepository repo)
{
    // Map input to TaskItem and enforce UTC for dates
    var task = new TaskItem
    {
        Title = input.Title,
        Description = input.Description,
        Owner = input.Owner,
        Created = input.Created.ToUniversalTime(),
        DueDate = input.DueDate.ToUniversalTime(),
        Status = input.Status,
        Comments = new List<Comment>()
    };

    // Save task in repository
    return await repo.AddTaskAsync(task);
}
```

Figure 18. GraphQL Mutation `AddTask` in the `Mutation` class in the backend.

For all GraphQL mutations and queries, separate input types, called data transfer objects (DTOs), are used. In general, it is not always recommended to directly expose EF Core entities, such as the `TaskItem` class, directly as a GraphQL input type. This may occur because certain properties should remain hidden from the client or are excluded from the mutation (Microsoft, Create Data Transfer Objects (DTOs), 2022). In this application, the list of comments is not supplied as an input of the mutation when a new task is created because a new task gets assigned a new and empty list of comments per default. As a result, as seen in Figure 19, the input DTO for adding a new task does not include the list of comments. Since SQLite automatically assigns a unique ID to the task when stored in the database, it is not necessary to supply it as an input when creating a new task. Consequently, the ID does not show up in the Data Transfer Object `GraphQLAddTaskInput`.

```
public class GraphQLAddTaskInput
{
    public string Title { get; set; }
    public string Description { get; set; }
    public string Owner { get; set; }
    public DateTime Created { get; set; }
    public DateTime DueDate { get; set; }
    public TaskItemStatus Status { get; set; }
}
```

Figure 19. Data Transfer Object `GraphQLAddTaskInput` in the class library.

To use GraphQL and the GraphQL server in the backend it is necessary to add the GraphQL server as well as the query- and mutation type (the respective classes) to the application builder in the `Program.cs` of the backend.

Figure 20 shows an example mutation to add a new task using Postman. In addition to the input, the desired output of the mutation is specified.

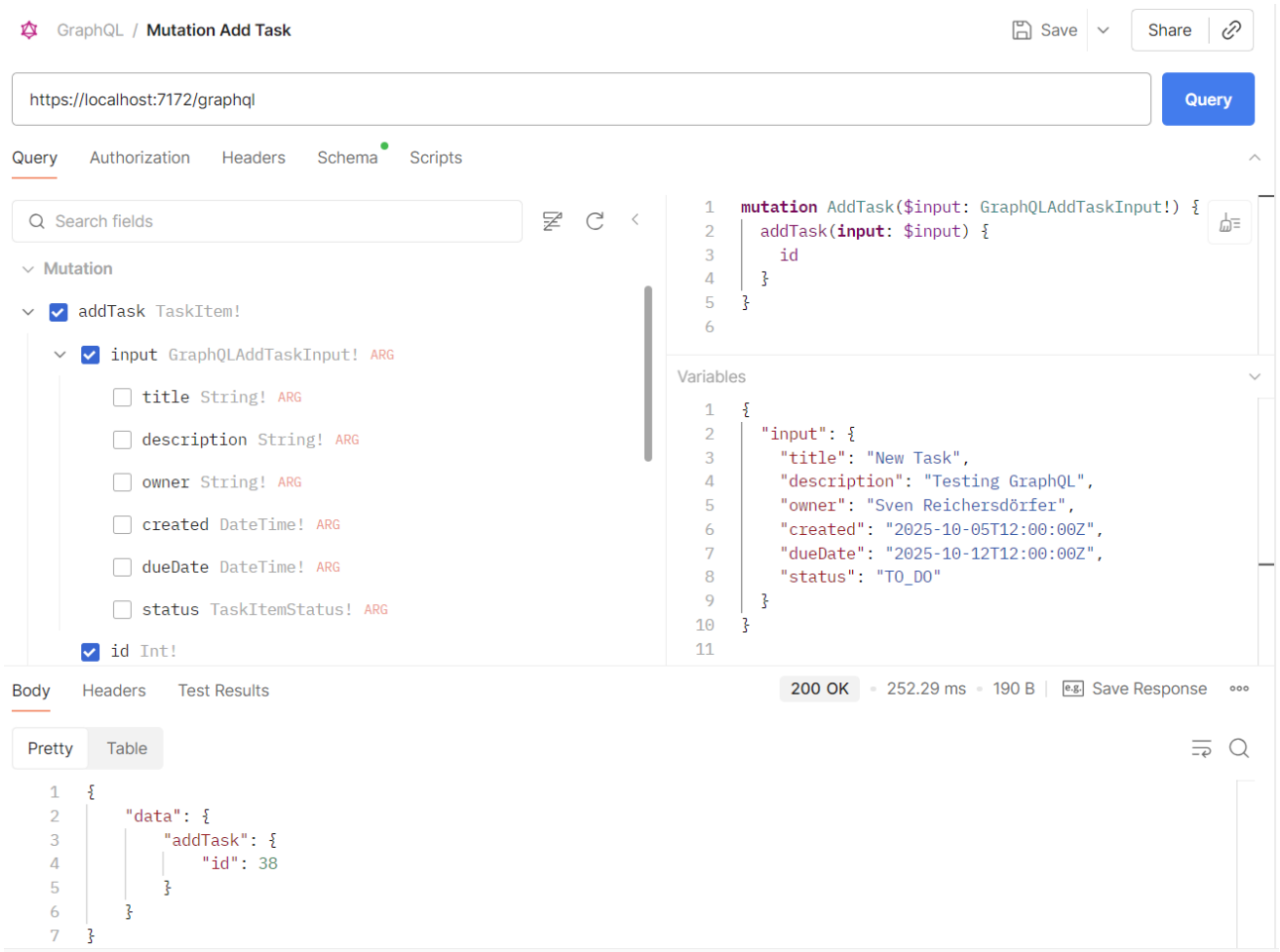

Figure 20. Example Mutation `AddTask` in Postman.

The line `app.MapGraphQL("/graphql")` in the `Program.cs` registers the GraphQL endpoint in the backend and defines the route. This way, queries and mutation can be executed with the URL of the server plus the `/graphql` suffix indicating the route.

4.3.4 gRPC services

The backend of the application acts as a gRPC server. The services supplied via the gRPC API are defined in the `TaskServiceGrpc.cs` file in the project. One major difference from the other APIs is that gRPC uses `.proto` files as described in Chapter 3.4. The `.proto` file contains the service definition such as `GetTask`, `AddTask`, `DeleteTask` and such, as well as the input and output of each service, called `message`. Figure 21 shows the definition of the `GetTask` service with the corresponding input `TaskRequest` and reply `TaskReply`. The `message` defines the datatype as well as the `field numbers` which are unique identifiers for each field in a `.proto` message (Protocol Buffer Team, n.d.).

```
// Reply for a single task
message TaskReply {
  int32 id = 1;
  string title = 2;
  string description = 3;
  string owner = 4;
  google.protobuf.Timestamp created = 5;
  google.protobuf.Timestamp dueDate = 6;
  TaskStatus status = 7;

  // All comments of this task
  repeated CommentReply comments = 8;
}
```

```
// Service for managing tasks
service TaskService {
    // Get a single task by ID
    rpc GetTask (TaskRequest)
     returns (TaskReply);
}

// Request a task by ID
message TaskRequest {
    int32 id = 1; // Task ID
}
```

Figure 21. Extract of `.proto` file for querying a single task with gRPC.

The reply when, for example reading a single task, has the type `TaskReply` thus must be mapped to a new `TaskItem` which is the type used in the application for tasks. This is done in a separate method in the `TaskServiceGrpc` class.

The gRPC `TaskService` class is registered with the line `app.MapGrpcService<TaskService-Grpc>()` in the `Program.cs` file in the backend. Furthermore, the gRPC service is assigned to the builder, as seen in Figure 22, and options such as less detailed error messages and a compression of the response are passed. The following Chapter 4.3.5 describes in more detail how the `.proto` file is registered in the class library to be used in the front- and backend.

```
// Add gRPC services
builder.Services.AddGrpc(options =>
{
    options.EnableDetailedErrors = false;
    options.ResponseCompressionAlgorithm = "gzip";
    options.ResponseCompressionLevel =
        System.IO.Compression.CompressionLevel.Fastest;
});
```

Figure 22. Adding gRPC services to the app builder.

4.3.5 Class Library

The class library is used to share methods or pieces of code between the front- and backend to avoid redundant code and ensure consistency. It contains the data structures stored in the SQLite database as well as helper methods to convert the status of a task or the time it was created to the right format. Furthermore, methods to map a GraphQL reply to as task or comment are needed in the backend and the frontend. Additionally, the DTOs mentioned in Chapter 4.3.3 are in the class library.

The `.proto` file, which is also contained in the class library, is referenced in the project file of the class library, so that the necessary code is generated for both the client and the server. Figure 23 shows how the `.proto` file is included in the `clib.csproj` file.

```
<ItemGroup>
    <Protobuf Include="Protos\TaskService.proto" GrpcServices="Both" />
</ItemGroup>
```

Figure 23. Extract of the `clib.csproj` file of the Class Library referencing the `.proto` file

This way, when the class library is referenced in the application, it already contains the necessary gRPC code.

4.3.6 Database Access

To access the SQLite database a repository for the tasks and the comments is used in the project. The independent repository provides clear separation of concerns by keeping the data logic out of the controllers and services of the APIs and additionally allows for easier testing and greater flexibility for maintaining and extending the data access logic.

The `TasksContext` class acts as the bridge between the application and the database, managing data access through Entity Framework Core. It defines `DbSet<TaskItem>` and `DbSet<Comment>` properties, which EF Core uses to automatically create corresponding tables based on the class models. This way no tables must be created by the user and using migrations, the initial database schema is generated and kept in sync with model changes. As a result, EF Core handles table creation, updates, and relationships automatically. An initial migration and application of the migration, for example in the developer console, is needed to create the dataset.

The following Figure 24 shows an extract from the `TasksContext` class. In the class the `DbSet` properties are defined and the path to the database file is assigned.

```
/// <summary>
/// Entity Framework Core DbContext for managing TaskItems and Comments.
/// </summary>
public class TasksContext : DbContext
{
    // DbSet representing the collection of TaskItem entities
    public DbSet<TaskItem> TaskItems { get; set; } = null!;

    // DbSet representing the collection of Comment entities
    public DbSet<Comment> Comments { get; set; }

    // Path to the SQLite database file
    public string DbPath { get; }

    public TasksContext(DbContextOptions<TasksContext> options)
        : base(options) { }

    public TasksContext() { }

    protected override void OnConfiguring
        (DbContextOptionsBuilder optionsBuilder)
            => optionsBuilder.UseSqlite($"Data Source=database.db");
```

Figure 24. Extract of `TasksContext` class managing tasks and comments in the backend.

In the `TaskRepository` class, partly depicted in Figure 25, a new `TaskContext` can be created which can then be used to for example read all `TaskItems` including comments as it is done in the method `GetAllTasksAsync`. In the same way `TaskItems` or `Comments` can be manipulated, added to or removed from the SQLite database.

```
public class TaskRepository
{
private readonly TasksContext _context;

public TaskRepository(TasksContext context)
{
    _context = context;
}

public async Task<List<TaskItem>> GetAllTasksAsync()
{
    return await _context.TaskItems.AsNoTracking()
            .Include(t => t.Comments)    // load related comments
            .ToListAsync();
        }
```

Figure 25. Extract of `TaskRepository` and `GetAllTasksAsync` Method

Like the REST controllers and the GraphQL server, the database must be added to the builder as `DbContext` before the app is built in the `Program.cs` of the backend.

4.4 Frontend implementation with Blazor Server

4.4.1 User Interface

The start page of the frontend displays the `TaskItems` inside the three different boxes representing the status of the Task. Figure 26 shows the start page (called `TaskBoard` in this project) in the web browser.

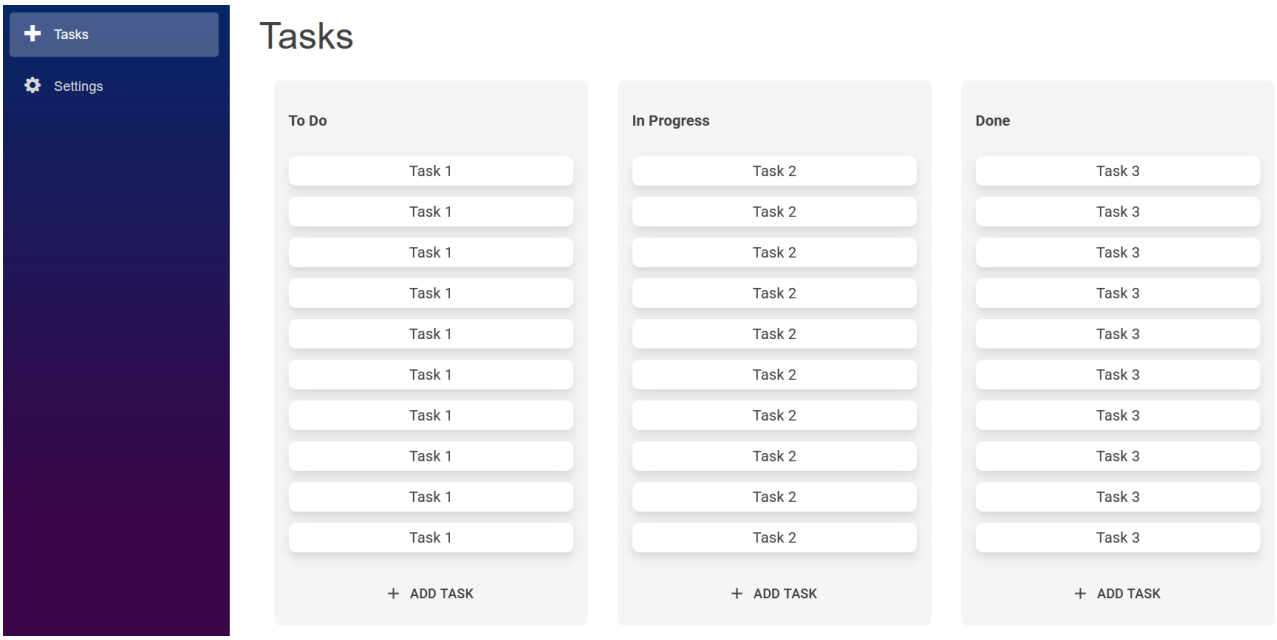

Figure 26. Screenshot of the start page of the frontend in Blazor Server (`TaskBoard`).

Each task can be moved to a different box by drag and drop and new tasks can be added by clicking the `ADD TASK` button and providing a title for the new task. Furthermore, the `TaskDetailsDialog` can be opened by clicking on the respective task.

Figure 27 shows the `TaskDetailsDialog` in which the details of the task, including title, description, owner, status and due date can be edited. Additionally, the dialog allows the user to add and delete comments or delete the entire task, by clicking the trash icon in the upper right corner.

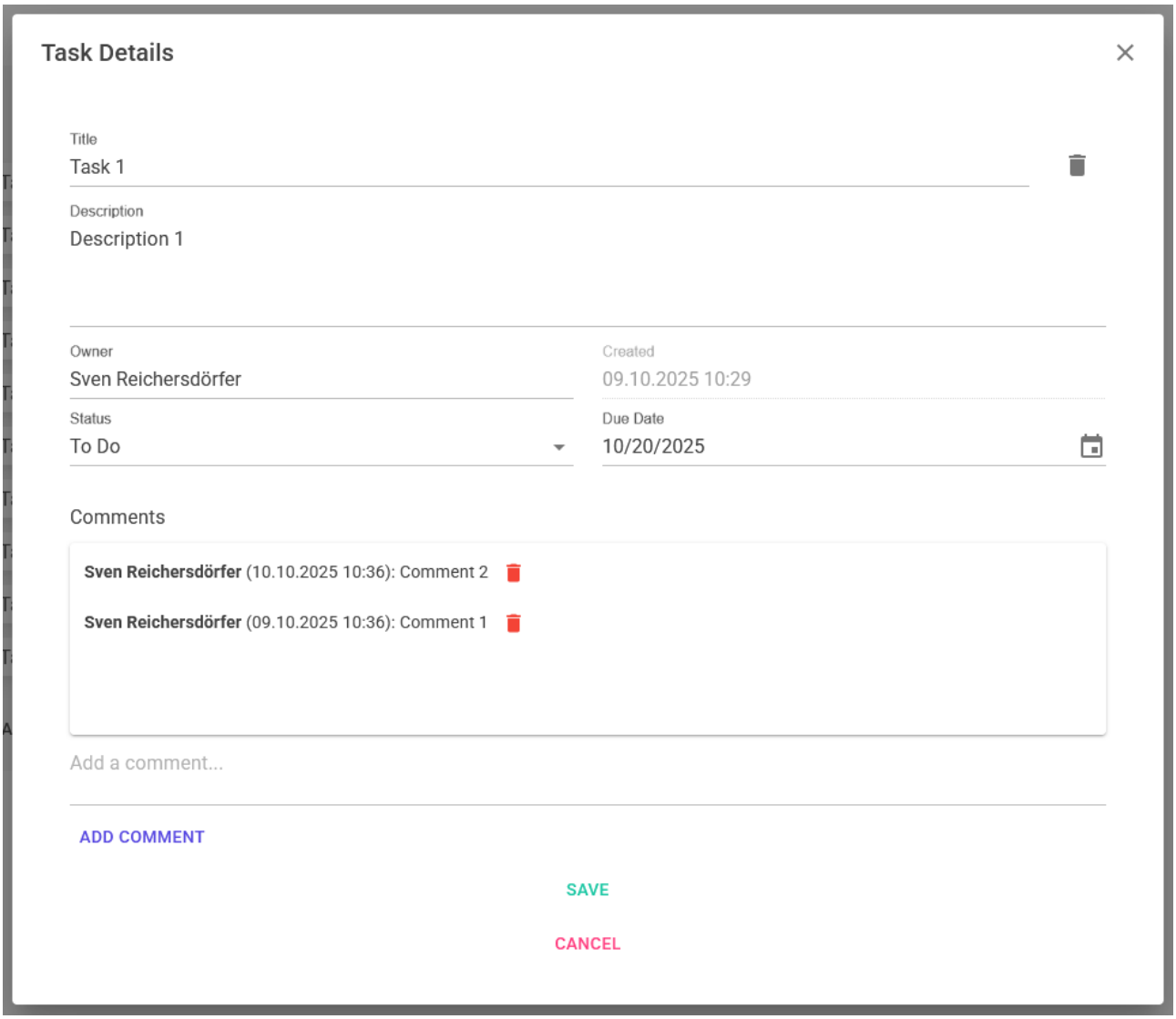

Figure 27. Dialog `TaskDetailsDialog` displaying details of the selected task.

4.4.2 Implementation of the User Interface

The frontend consists of three pages: `TaskBoard`, `TaskDetailsDialog`, and `Settings`. The two pages `TaskBoard` and `Settings` must be registered in the `NavMenu.razor` and both get a `@page` value assigned at the top of the razor file. The `TaskDetailsDialog` is opened as a dialog, meaning a separate window opens on top of the `TaskBoard`, so the details page of each task does not show up in the navigation menu on the left side of the website thus does not have to be in the `NavMenu.razor`.

The `TaskBoard` page is also the page that is navigated to when the application is started, when the settings are saved or the opened task is closed or deleted. The navigation between pages inside the application is straightforward and most of the tasks are automatically handled by MudBlazor. To navigate from the `Settings` page to the `TaskBoard`, the navigation manager, injected at the beginning of the code of the page, can be called with the `NavigateTo` method and the respective `@page` value of the `TaskBoard` page.

As described in Chapter 2.2.5, MudBlazor is used to create the UI and communicate with the backend. MudBlazor is then included in each razor file with the `@using` keyword. This way, the MudBlazor components are available to be used in the razor component.

The `TaskBoard` page consists of a `DropContainer` that displays the three different boxes ("sections") for the respective task status. Each section is created by using a `MudPaper` component in which all tasks are displayed. Furthermore, each section contains the `AddTask` button which, when clicked, reveals a `MudTextField` to enter the name of the new task.

The `TaskDetailsDialog` displays all details of the selected task in a popup window that opens on top of the `TaskBoard`. All the code of the page is inside a `MudDialog` in the `TaskDetailsDialog.razor` file and the dialog can be opened by calling the `DialogService.Show` method in the `TaskBoard.razor` component. Additionally, parameters and options can be handed to the `DialogService` in the `.Show` method. In this application the id of the selected task as well as options like the maximum width of the dialog, the decision to show a close button, or whether to close the dialog when the escape key is pressed, is handed to the method. The dialog itself consists of simple text boxes and a date picker for the due date. The comments are displayed in a `MudStack` and each comment entity gets assigned a corresponding icon button which can be used to delete the comment.

Figure 28 shows the source code for creating the comment box in the `TaskDetailDialog`, adding comments to the box and adding a delete button to each comment.

```
<MudPaper Elevation="1" Class="pa-2" Style="height: 160px; overflow-y: auto;">
    <MudStack Spacing="2">
        @if (Task.Comments != null && Task.Comments.Any())
        {
            @foreach (var comment in Task.Comments.OrderByDescending(
                c => c.Date))
            {
                <MudPaper Elevation="0" Class="pa-1 d-flex align-center
                    justify-between">
                    @*Show author, date and content of comment*@
                    <MudText Typo="Typo.body2" Class="mr-2">
                        <b>@comment.Author</b>
                        (@clib.Helper.TimeHelper.ToTimeZoneString(
                            comment.Date,
                            SettingsService.Settings.TimeZoneId)):
                                @comment.Text
                    </MudText>

                    @*Delete button behind each comment entry*@
                    <MudIconButton Icon="@Icons.Material.Filled.Delete"
                                   Color="Color.Error"
                                   Size="Size.Small"
                                   OnClick="@(() =>
                                       DeleteComment(comment))" />
                </MudPaper>
            }
        }
        else
        {
            <MudText Typo="Typo.body2">No comments yet.</MudText>
        }
    </MudStack>
</MudPaper>
```

Figure 28. MudBlazor code for comments in `TaskDetailDialog` in the frontend.

4.5 Integration and API consumption by the frontend

Each API is consumed in different ways by the frontend. For every action in the respective razor page, one Method differentiates between the selected API and calls the respective method.

REST uses a client that is created by calling `ClientFactory.CreateClient("backend")` in every REST method in the frontend. The backend is defined as the local host in the `Program.cs` of the frontend. Depending on which HTTP method is used, the client can be used to for example asynchronously call `client.GetFromJsonAsync<List<clib.Models.TaskItem>>("TaskItem")` which retrieves all tasks from the backend respectively the database. The string "`TaskItem`" , supplied as a parameter, defines the REST endpoint displayed in Figure 17. The data type `List<clib.Model.TaskItem>` of the returned value also shows that each task in the reply

is automatically mapped to the `TaskItem` class. To update a single task, the client can also be used to call a PUT method, and the updated tasks can be passed in the `client.PutAsJsonAsync` method.

While GraphQL uses the same client and backend, it additionally requires the GraphQL request which is always sent to the backend via a HTTP POST method. The response from the backend is then stored in a variable and the information can be converted to JSON by calling `var gqlResponse = await response.Content.ReadFromJsonAsync<clib.DTOs.GraphQLTasksResponse>()`. As the line of code suggests, the same DTOs out of the class library are also used in the frontend implementation. For every task in `GraphQLTasksResponse`, a new task with the type `TaskItem` is created. A similar approach is used in all other query type GraphQL requests. For mutations, for example to update or delete a task, the response is usually of less interest thus only the status code of the response is checked. The status code of the response can be checked by calling `response.IsSuccessStatusCode` which returns a Boolean value signaling the success of failure of the mutation. All required input data for the mutation can be added to the request before sending it. Mutations are sent in the same way as queries, by calling the PUT method with the backend client.

For gRPC, client-side logic is implemented which interacts with the gRPC server in the backend. The service contains all methods to perform the CRUD operations on tasks and comments. The methods called by the client, which is also created in the service, are defined in the `.proto` file that is used by the front- and backend. The service also includes several mapping methods that map responses to actual `TaskItem` objects or vice versa maps objects of type `TaskItem` to types required for the gRPC message. With the service class and the `.proto` file, the sending of a gRPC request is rather straightforward and can be done by creating a respective request, depending on which action ought to be executed. With the request and the client created by injecting the service class into the razor page, the method contained in the service class can be called, and the request passed to the method.

Figure 29 shows the `LoadTasksGrpc` method where the injected client `GrpcClient` is used to call `GetAllTasksAsync` method, located in the gRPC service class, which fetches all tasks from the gRPC service asynchronously.

```
private async Task LoadTasksGrpc()
{
    // Read all TaskItems
    var reply = await GrpcClient.GetAllTasksAsync(
                        new clib.Protos.Empty());

    if (reply.Tasks != null && reply.Tasks.Count > 0)
    {
        // Create new TaskItem for each task
        var taskItems = reply.Tasks.Select(grpcTask =>
                            new clib.Models.TaskItem
        {
            Id = grpcTask.Id,
            Title = grpcTask.Title,
            Description = grpcTask.Description,
            Owner = grpcTask.Owner,
            Created = grpcTask.Created != null ?
                        grpcTask.Created.ToDateTime() :
                            (DateTime?)null,
            DueDate = grpcTask.DueDate != null ?
                        grpcTask.DueDate.ToDateTime() :
                            (DateTime?)null,
            Status = (clib.Models.TaskItemStatus)grpcTask.Status,
            // Add comments to task if it has any if not add empty list
            Comments = grpcTask.Comments?.Select(c =>
                        new clib.Models.Comment
               {
                   Id = c.Id,
                   TaskItemId = c.TaskItemId,
                   Author = c.Author,
                   Text = c.Text,
                   Date = c.Date?.ToDateTime()
               }).ToList() ?? new List<clib.Models.Comment>()
            }).ToList();

            // Set task service to retrieved tasks
            TaskState.SetTasks(taskItems);
        }
    }
}
```

Figure 29. `LoadTasksGrpc` method in the frontend.

The code demonstrated the necessary conversion of Protobuf-specific formats, such as `Timestamp`, back into standard .NET types such as `DateTime`. Furthermore, the nested `Comments` are added to the `Task`, if the `Task` contains any, or alternatively a new `List` of `Comments` is assigned.

4.6 Application Settings

The frontend also includes a settings page that allows the selections of the current user, desired API type, application theme, default values for new tasks and time zone. The settings page is displayed in Figure 30.

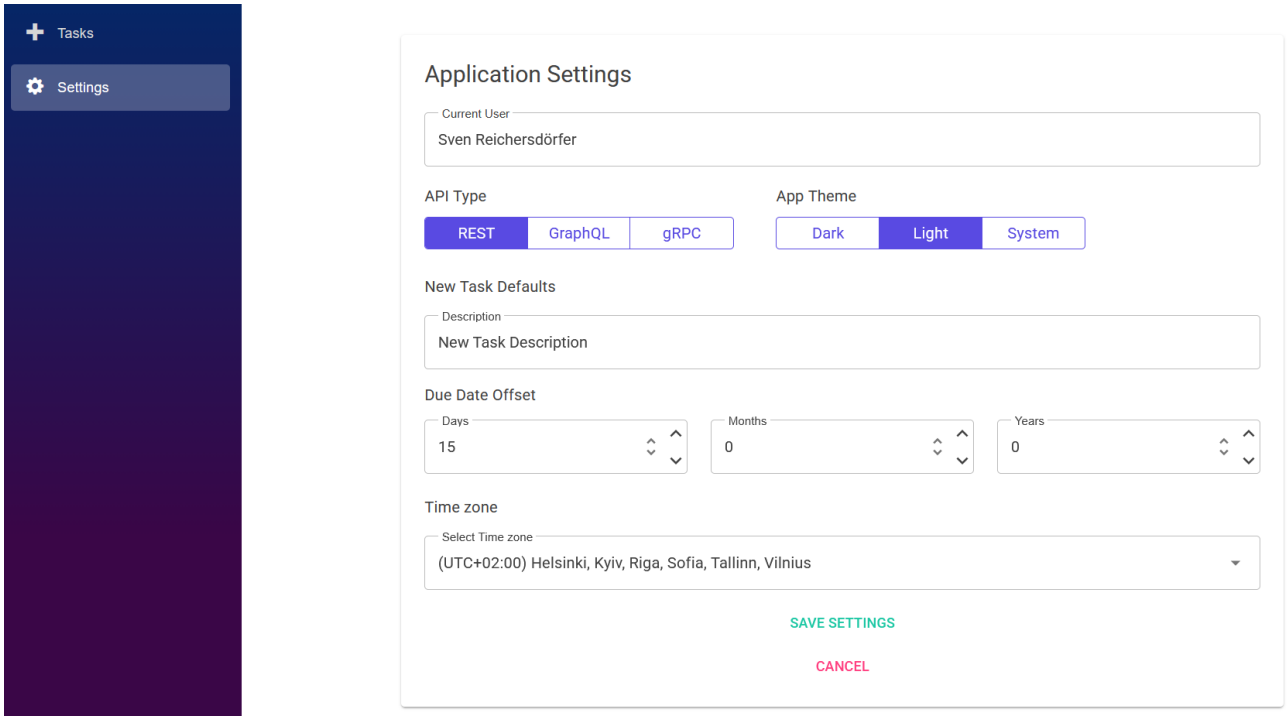

Figure 30. Screenshot of the Application Settings page.

The settings page contains, other than simple labels and text fields for the current user and the default task description, two `MudToggleGroups` with each three `MudToggleItems` to pick the API type and the color theme of the application. This type of control functions analogous to common radio buttons. Furthermore, a dropdown menu (`MudSelect`) allows to select the time zone and three `MudNumericFields`, contained in a `MudGrid`, the selection of the offset which is added to the date the new task was created, thus resembling the due date of the new task.

The settings are saved in a separate JSON file so that the previously made settings remain even when the application is closed. To save and load the settings in the JSON file, a `FileSettingsService` is used. The service can be injected into all razor files respectively razor components.

As soon as the application is started, the constructor of the service is called and the settings are loaded from the JSON file, which is stored on the computer. Figure 31 shows the content of the JSON file.


```
{
  "CurrentUser": "Sven Reichersd\u00F6rfer",
  "SelectedApi": "Rest",
  "NewTask": {
    "Id": 0,
    "Title": "",
    "Description": "New Task Description",
    "Owner": "",
    "Created": "2025-09-22T14:20:02.1067232+03:00",
    "DueDate": "2025-10-21T00:00:00",
    "Status": "ToDo",
    "Comments": []
  },
  "DueDateDaysOffset": 15,
  "DueDateMonthsOffset": 0,
  "DueDateYearsOffset": 0,
  "TimeZoneId": "FLE Standard Time",
  "AppTheme": "Light"
}
```


Figure 31. Content of the Settings JSON file.

The use of the `System.Text.Json` namespace in the `FileSettingsService` allows the use of the `JsonSerializer`, which automatically serializes and deserializes the `AppSettings` object when writing to or reading from the JSON file.

4.7 Testing setup

4.7.1 Software Components of Testing Setup

The software setup to test and benchmark the backend consists of four components as described in the following paragraphs.

Postman is used to test the three APIs without the need for a frontend. This way the backend can be developed and implemented first. Postman allows to save queries, mutations, requests, and so forth, depending on the type of API, in so called collections. The type of API must be selected when creating a new collection. The URL and an endpoint are the only necessary information for

sending a query via REST. For GraphQL, the schema structure is loaded from the API and displayed in Postman, to easily edit the input variables or desired data for the reply. For gRPC, the `.proto` file can be imported into Postman to use the services defined in the file.

DB Browser for SQLite is used to check the data structure of the SQLite database and the actual data in the database. The DB Browser clearly depicts the data structure of TaskItems and Comments. Furthermore, DB Browser allows you to view the data in the database and edit individual cells which helps with testing and developing the database access in the application. Figure 32 shows a screenshot out of the DB Browser. In the center of the image, the data content of the database is depicted. The bottom right corner of the image shows the schema of the database and above that, the field for editing individual cells in the database is located.

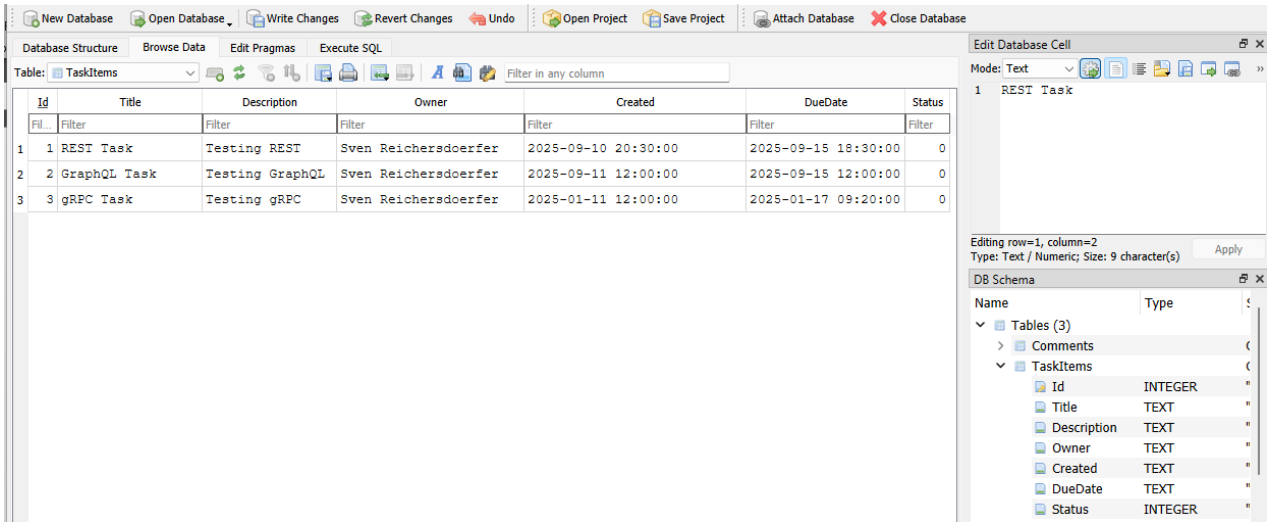

Figure 32. Screenshot of the DB Browser showing the `TaskItems` in the database.

NLog is a library for .NET application which is used to easily log error messages, instructions or other messages in customizable formats. In this application, only the basic functions of NLog are used to log when, for example, a new request is received, or the database is accessed. A log in this case means an output via the console in Visual Studio. The configuration of NLog is done via the `NLog.config` file which is located in the project folder.

Grafana k6 is the main tool to benchmark the backend. It allows the user to write tests in JavaScript and use them to simulate requests to test a website or API. Once installed, k6 can be run via the console by calling k6 and the respective JavaScript file (Moradin, 2019).

To test the backend three different scenarios are applied:

- `small_query`: Describes a scenario where a single task is fetched. This scenario can be used as a baseline for simple requests. It could for example occur when the application is started and all task titles are displayed on the start page, but not all task details or comments need to be retrieved.
- `large_query`: Retrieves a full task, including all comments and nested data to simulate heavier payloads and evaluate potential serialization and deserialization overhead. This type of query is used when a task is selected, and all task details and comments are displayed on the frontend.
- `concurrent_load`: Simulates multiple virtual users executing requests simultaneously to assess scalability, throughput and server resource utilization under load. This type of load could occur when the application is shared with multiple users that can create, edit or delete tasks at the same time.

The benchmarking tool k6 allows to create different scenarios and accompanying test functions in a JavaScript file. Figure 33 shows the definition of the `small_query` scenario. As the comments in the code suggest, the function to be executed, the number of users as well as the test duration is stated as options for the scenario.

```
export let options = {
    scenarios: {
        small_query: {
            executor: 'constant-vus',  // run constant number of vus
            exec: 'smallQuery',        // the JS function to execute
            vus: 1,                    // number of virtual users
            duration: '10s',           // test runs for 10 seconds
        },
```

Figure 33. Definition of the `small_query` testing scenario in JavaScript for k6.

Since the `small_query` and `large_query` do not use multiple users at the same time, the amount of virtual users (`vus`) is set to one constant user. Furthermore, the test is run for 10 seconds, as stated in behind the `duration` option.

Once the scenario is defined, the respective function, called in the scenario, is added to the same JavaScript file. The following Figure 34 shows the function `smallQuery` for REST, used to continuously fetch a single task for 10 seconds.

```
export function smallQuery() {
    let res = http.get(`${BASE_URL}/109`); // Fetch one task by ID
    // Validate response using checks
    check(res, {
        'status is 200': (r) => r.status === 200,
        'latency < 200ms': (r) => r.timings.duration < 200,
    });
    sleep(1); // Wait 1 second between iterations to simulate user pacing
}
```

Figure 34. Definition of the `smallQuery` testing function for REST.

In the function, `http.get` is called with the URL of the server / endpoint. Also, a fixed `ID` of a task known to exist is provided in the `GET` method. The function waits one second in between every iteration to simulate real user pacing. Furthermore, the response is validated by checking the response status, status 200 meaning the query was successful. The `smallQuery` function additionally checks if the latency is below 200 milliseconds. The result of this check, however, is not considered in this analysis.

The testing functions for gRPC and GraphQL are similar. For GraphQL, a query string is defined and send to the http endpoint via a POST method. To test gRPC, a client needs to be defined like and the `.proto` file loaded. Then, the gRPC services in the `.proto` file can be used. Figure 35 shows the definition of the `smallQuery` testing function for gRPC and how the `client` uses the `TaskService` methods defined in the `.proto` file to fetch one single `Task`.

```
export function smallQuery() {
    ensureConnected(); // Helper method to ensure established connection
    const res = client.invoke('TaskService/GetTask', { 109 }, { timeout: '3s' });
    check(res, { 'status OK': (r) => r && r.status === grpc.StatusOK });
    sleep(1); // Wait 1 second between iterations to simulate user pacing
}
```

Figure 35. Definition of `smallQuery` testing function for gRPC.

4.7.2 Hardware of Testing Setup

To make the results of this thesis more comparable, Table 5 shows a list of the hard- and software components used as well as referenced libraries in the source code.

Table 5. Used Soft- and Hardware of the testing setup.

Software and Libraries	**Hardware**
Google Protobuf 3.32.1 Grpc.AspNetCore 2.71.0 HotChocolate.AspNetCore 15.1.10 MudBlazor 7.16.0 NLog 6.0.4 Microsoft.EntityFrameworkCore 9.0.9 Web_api (Console Application in .NET 8.0) Clib (Class Library in .NET 7.0) Blazor (Blazor Server in .NET 7.0) Visual Studio 17.14.16	CPU: 13th Gen Intel(R) Core(TM) i7-1355U 1.70 GHz RAM: 32.0 GB OS: Windows 11 Pro 23H2 Graphics: Intel(R) Iris(R) Xe Graphics

5 Results

The purpose of this chapter is to present the findings, compare them, and interpret the results. This chapter ends with a summary as well as recommendations for the practice.

5.1 Performance evaluation results

5.1.1 Latency and Response Time

To measure one key component of the performance of the APIs, the latency of each API was tested with three different scenarios. The scenarios are described in more detail in the preceding Chapter 4.7. For the tests, the database contained 30 tasks (10 tasks in each status), each task containing two corresponding comments.

Figure 36 shows the average latency per API type and scenario. The latency describes the time from the start of the request to the end of the response.

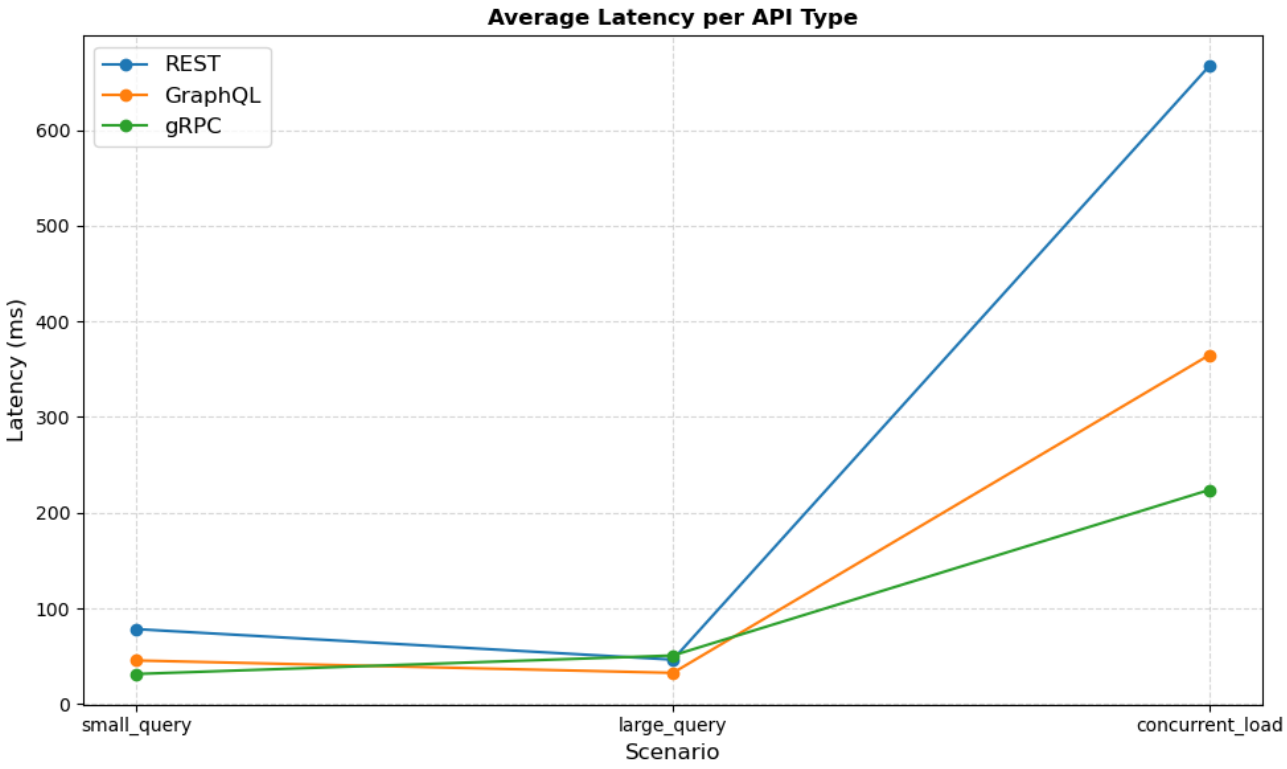

Figure 36. Average Latency per API type and Scenario.

While gRPC showed a steady rise in latency for an increasing amount of load, REST and GraphQL showed a slightly lower latency for the large query. Possible reasons for this result include the fact

that large queries might reuse cached database results or warmed-up connections (Newton-King, Performance best practices with gRPC, 2025). Moreover, for short-lived connections more connection overhead needs to be handled, dominating the noticeably short payload. For the larger queries, the data transmission time becomes a bigger part of the total latency which allows the latency to be lower compared to the smaller queries. Under concurrent load, gRPC registered the lowest latency of on average 0.223 seconds, while REST was clearly least suited for concurrent load, showing a high average latency of more than 0.65 seconds.

Another metric used to compare the API performance is the iteration duration. Each iteration corresponds to one full test loop per virtual user (VU). The iteration duration measures how long each iteration took (including think/sleep time). An average value of 0.457 seconds for gRPC means, on average, each virtual user completed a gRPC request every 0.457 seconds. The total time per virtual user iteration indicates the APIs responsiveness under load. Figure 37 depicts the iteration duration for all three APIs.

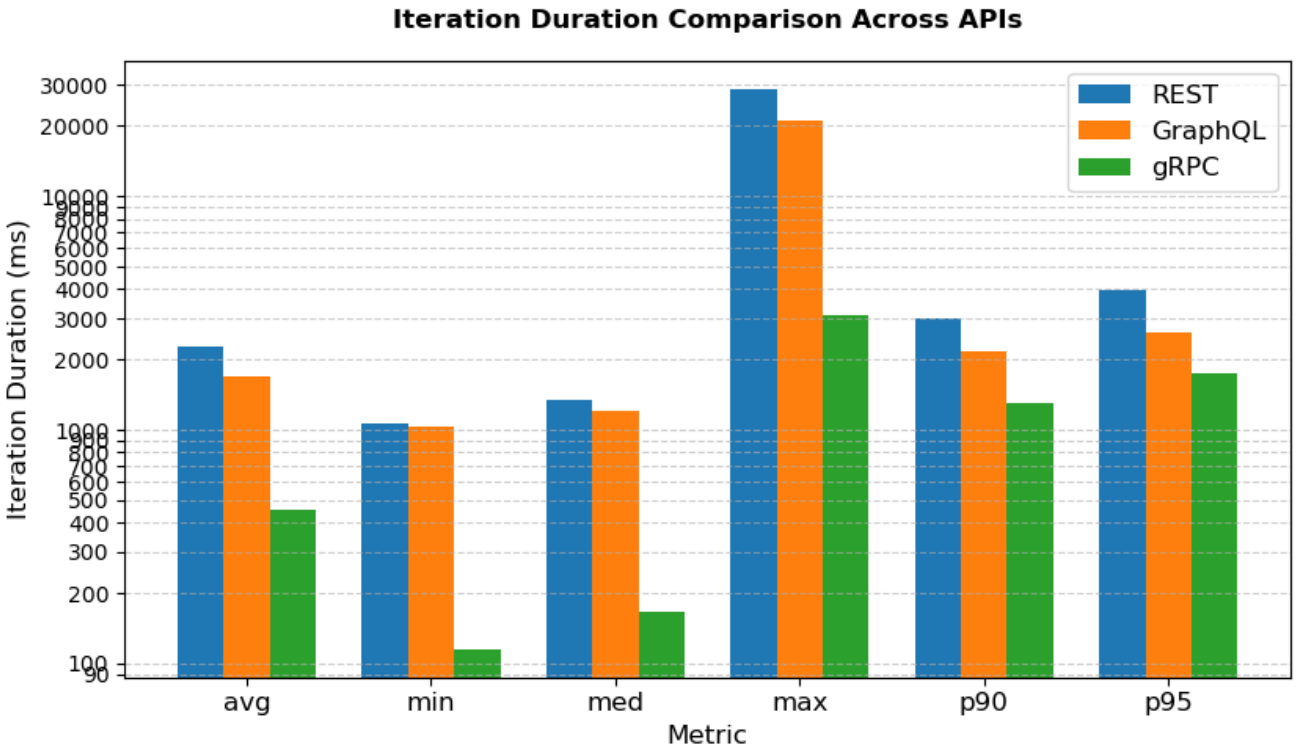

Figure 37. Iteration Duration comparison across APIs.

The three left most columns show the average iteration duration for each API. Furthermore, the minimum, median and maximum values are displayed. The p90 and p95 values indicate that 90% respectively 95% of the iterations took less time than these values. These two values are useful to show performance consistency and detect outliers in the iteration duration.

Comparing the three APIs amongst each other, gRPC had the shortest iteration duration with an average of only 0.457 seconds, while REST and GraphQL did not differ too much from each other (2.27 seconds and 1.7 seconds). The efficiency of gRPC can be a direct result of its binary message format and the high-speed protocol, which minimizes transport overhead (Loganathan, 2024). GraphQL's ability to consolidate multiple REST calls into a single request also reduces the overhead (Postman Team, gRPC vs. GraphQL, 2024) and querying and return only the necessary data furthermore reduces the amount of transferred data thus the iteration duration. REST had the longest average iteration duration of the three APIs. REST's tendency towards over-fetching data leading to higher payloads can be one cause for the higher iteration duration. The remarkably high max values of REST and GraphQL, but comparable close p90 / p95 values of all three APIs also suggest that for REST and GraphQL, infrequent high-latency requests or spikes appear, which gRPC seems to handle better.

5.1.2 Payload Size

To compare and analyze the payload size for each API the size of the request and response was recorded. Figure 38 shows the average request and response size per API and scenario.

As can be seen in Figure 38, for small queries REST and GraphQL send and receive similar amounts of data per request, with 507 bytes for GraphQL and 477 bytes for REST in total. gRPC, on the other hand, transfers significantly smaller payloads of only around 120 bytes in total.

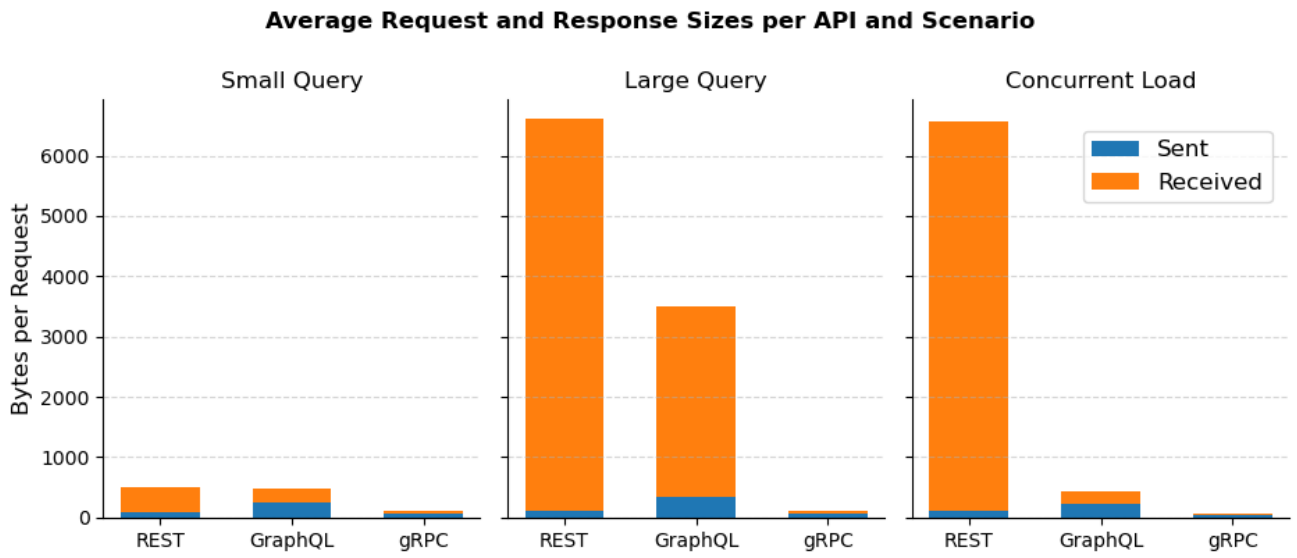

Figure 38. Average Request and Response sizes per API and Scenario.

For larger queries, the REST endpoint returns the largest amount of payload with 6602 bytes, indicating a potential over-fetching. GraphQL, in comparison, shows significantly smaller payload of 3506 bytes, due to the more efficient fetches. Again, gRPC, with its highly compact binary format, shows by far the smallest payload with virtually the same value of approximately 120 bytes.

Under concurrent load the APIs also exhibit notable differences in the per-request payload size. REST transfers the most substantial amounts of data per request with around 6.6 kB. While sending only small request payloads, the response payload is much larger for REST, which can impact network utilization under high concurrency. With approximately 0.43 kB, GraphQL maintains a smaller per-request payload, indicating its efficiency in fetching only the data that is truly needed. The overall payload of gRPC is by far the smallest with approximately only 78 bytes average per request.

5.1.3 Throughput and Resource Utilization

To analyze the throughput the following calculation can be applied to the data retrieved from the performance tests with k6:

$$Throughput = \frac{Total \text{ Requests}}{Test\ Duration(s)}.$$

The throughput indicates how well the server handles parallel requests, the way they occurred in the concurrent load test. It is measured in units of data per unit of time, which in this case is requests per second. Moreover, it is inversely related to latency and payload size and directly affects how many requests an API can handle per second. Figure 39 shows the calculated throughput of each API for the three scenarios.

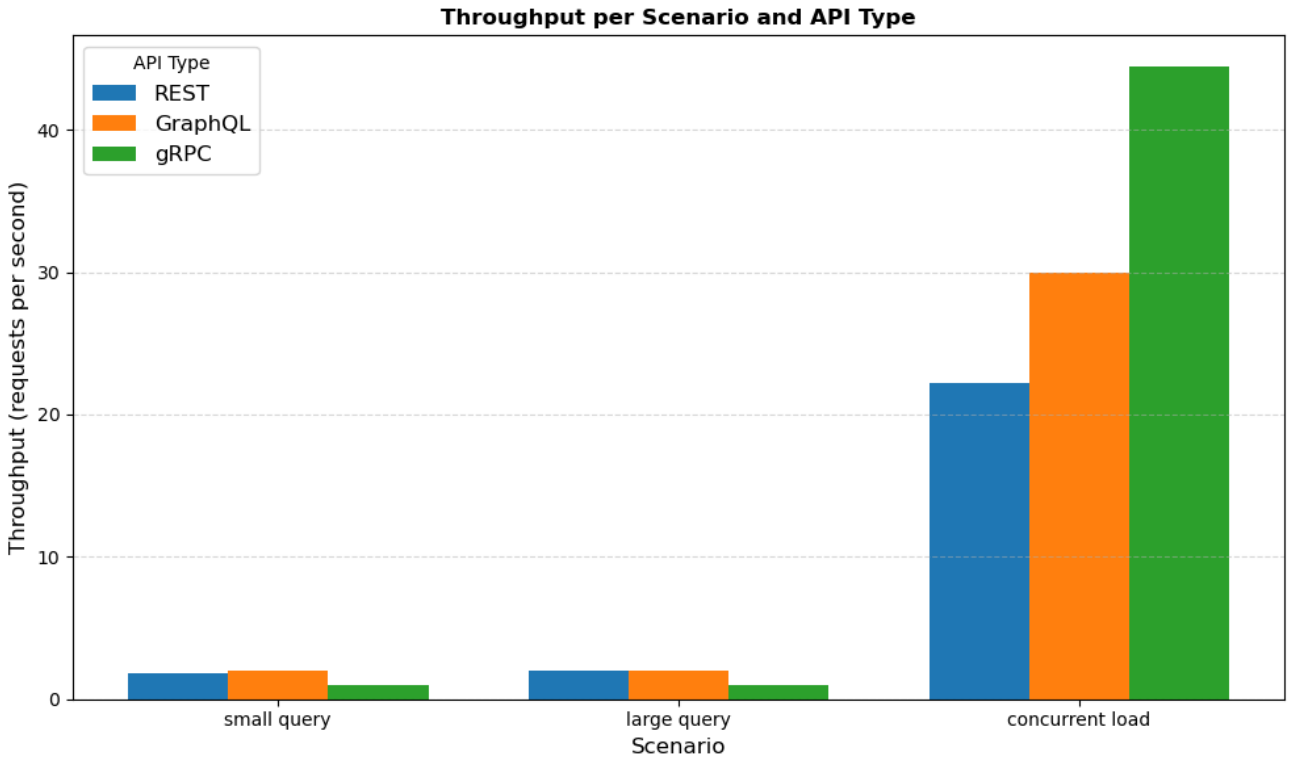

Figure 39. Throughput per Scenario and API type.

For small queries GraphQL achieved the highest throughput with 2 req/s (requests per second), followed by REST and gRPC, with around 1.8 req/s and 1 req/s. For larger queries both REST and GraphQL achieved a throughput of 2 req/s while gRPC dropped to 1 req/s. The lower throughput of gRPC for both the small and the large query could be explained by a larger overhead as well as the way the more complex HTTP/2 stream of gRPC is handled by k6.

gRPC's performance capabilities really show under concurrent load. It achieves by far the highest throughput of over 44 req/s followed by GraphQL with almost 30 req/s and REST with approximately 22 req/s. This is most likely achieved by gRPC's use of features like multiplexing and its efficient binary framing comprised in HTTP/2 (Vantroys, 2024). GraphQL's ability to handle concurrent requests more efficiently is also reflected in the combination of the moderate latency (see also Figure 36) and low payload size per request (see also Figure 38). REST is limited by higher latency and request and response size under load.

CPU and RAM usage help evaluate efficiency and scalability. Lower values indicate that the server can handle more requests before resources become constrained. Spikes in usage values can show potential bottlenecks under load. To evaluate the resource utilization, CPU and RAM load during

the k6 performance tests was analyzed. Table 6 shows the average and maximum CPU and RAM load.

Table 6. CPU and RAM load for k6 performance tests.

	CPU		RAM	
API	Avg (%)	Max (%)	Avg (%)	Max (%)
REST	4.62	9.50	39.27	39.40
GraphQL	6.61	14.00	39.99	40.20
gRPC	7.98	12.40	39.55	39.70

gRPC exhibits the highest average CPU usage with 7.98 %. The comparatively low CPU usage of REST with 4.62 % likely stems from the simpler request and response handling while the high spike of CPU use for GraphQL suggests that its dynamic query resolution can be more CPU-intensive under certain requests (Pérez, 2019). The average RAM usage is similar across all three APIs, indicating that memory footprint is less determined by the API style but rather the runtime environment and k6 load.

5.2 Developer Experience analysis

5.2.1 Ease of Implementation & Implementation Time

REST was the most straightforward to implement due to its minimal setup requirements and the available libraries for handling HTTP and JSON. GraphQL, on the other hand, required an additional schema definition in each query and mutation as well as data separate input types. While this allows for a more structured approach for data fetching, it also introduces complexity to the program. gRPC was the most complex and time-consuming API to set up. This was mainly due to the required protocol buffer definitions in the *.proto* file, as well as the necessity to map the reply to the `TaskItem` structure of the program.

In terms of extending the front- and backend and adding new features there were also distinct differences in the three APIs. Table 7 shows the result of measuring the time to add a new feature to both the front- and backend, which allows the query of the five most recent tasks with the provided status.

Table 7. Comparison of the Implementation Time for a new feature with all APIs.

API	Time Backend	Time Frontend	Total Time
REST	10 min	3 min	13 min
GraphQL	8 min	21 min	29 min
gRPC	12 min	4 min	16 min

The results show that adding new features, especially if they are similar to already existing features, does not require much work with all three APIs. However, there is a difference between the three APIs, especially in the frontend implementation time. While the backend is implemented in similar amounts of time for REST, GraphQL and gRPC, the frontend implementation of GraphQL takes significantly longer than for the other two APIs. One reason is that GraphQL requires new DTOs for the new query. Furthermore, creating the new query string as well as debugging took more time compared to REST and gRPC where the code for the new query was much more copy and paste work.

It is however important to consider that depending on the individual skillset and familiarity with the respective API technology, some developers might need significantly less time to implement a feature in GraphQL and more time for gRPC, for instance. Since the newly implemented feature was comparably easy and all three API technologies already fairly similar, the time in Table 7 can still roughly reflect the difference in implementation time for new features for the respective API.

Another metric that can be used to investigate the individual implementation complexity for each API is the amount of lines of code. Considered in the lines of code calculation are all non-comment and non-blank lines. This also includes variable declarations and statements. The following

Table 8 shows the files containing code for each API, separated into frontend and backend and the respective lines of code (LOC).

Table 8. Lines of code for each API in the respective file.

Layer	API	File	LOC
Frontend	REST	TaskRestService.cs	75
	GraphQL	TaskGraphqlService.cs	95
	gRPC	TaskGrpcService.cs + TaskGrpcMapper.cs	160
Backend	REST	TaskItemController.cs	97
		CommentController.cs	38
	GraphQL	Query.cs	27
		Mutation.cs	76
		DTOs/*.cs	35
		Helper/GraphQL.cs	45
	gRPC	TaskServiceGrpc.cs	185
		TaskService.proto	35

The lines of code for each API in Table 8 result in 210 lines for REST API, 278 lines for GraphQL and 380 lines for gRPC. gRPC needs the most amount of code, mainly due to the mapping layer (`TaskGrpcMapper`) and the stricter type handling. REST is more straightforward than GraphQL and gRPC and even though it requires separate endpoints for each operation, it requires less lines of code than the other two APIs. The code for GraphQL in parts out of the DTOs and the helper classes necessary to map data from the GraphQL response to the `TaskItem` class in the backend.

5.2.2 Learning Curve & Tooling Support

In terms of documentation and community resources, REST is by far the most established API style. It is supported by extensive documentation, tutorials, and community resources and nearly

all web frameworks provide built-in REST support, and many developers are already familiar with its principles. This maturity makes REST particularly easy to learn and adopt, even for beginners (Postman, State of the API Report, 2025).

GraphQL also benefits from strong community support, with extensive documentation available through open-source initiatives such as the GraphQL Foundation, Apollo, and various framework-specific libraries. However, the process of becoming familiar with GraphQL's model for representing and interacting with data, can take time. Developers must become proficient in fetching and manipulating data, handling frontend requests, and collaborating on a single GraphQL schema (Apollo, Apollo GraphQL, n.d.).

gRPC, in contrast, has a steeper learning curve. Developers must understand Protocol Buffers, service contracts, and streaming concepts, which are less common in typical web development. Although official documentation from Google and Microsoft is detailed, practical tutorials and beginner-friendly examples are less common, which can slow down onboarding. Furthermore, in some scenarios, workarounds and custom solutions are necessary because libraries or existing tools do not fully satisfy the developers' requirements (Vantroys, 2024).

Tooling support is a key factor in how efficiently developers can test, inspect, and debug APIs. REST benefits from a mature ecosystem and broad selection of tools such as Postman and Swagger (Nosowitz & Goodwin, gRPC vs. REST , n.d.). GraphQL also offers robust and growing tooling support with tools like GraphiQL, ApolloStudio or Prisma (Kundalia, 24). The tooling support for gRPC is still developing but has improved in the past years. gRPC now offers several useful tools such as BloomRPC, grpcurl as well as partial support in postman (Postman, State of the API Report, 2023).

5.2.3 Flexibility for Clients

When it comes to data exchange, REST typically relies on JSON, which offers flexibility but also weak type enforcement. This forces the client to handle type inconsistencies and perform type validation by themselves (Postman, State of the API Report, 2023). GraphQL, on the other hand, provides stronger type safety compared to REST, due to the predefined schema (Ray, 2021). The likelihood of runtime errors is further reduced by a constant validation of queries at runtime or during development through schema introspection and code generation (Krantz, 2023; Bonnin, 2024). By

using Protocol Buffers (Protobuf), gRPC enforces extraordinarily strong type safety. The message format and service contract is strictly defined in the `.proto` files reducing errors at runtime (Le, 2025).

REST endpoints return fixed data structure often leading to clients receiving more data than needed. Furthermore, occasionally multiple requests are necessary to collect related resources which could lead to over- or under-fetching. GraphQL allows clients to specify exactly which fields should be returned in a single query. This minimizes unnecessary data transfer as well as the number of requests required for more complex data structures. Due to the predefined data structure in the .proto file, clients cannot dynamically modify the response content at runtime. This leads to more efficiency but also causes a lack of flexibility.

5.3 Maintainability and Scalability analysis

5.3.1 Versioning and Evolution

Versioning in REST is commonly done by versioning URLs (e.g., `/api/v1`) or by headers. While this method is straightforward, it can lead to maintenance overhead or duplicated code when multiple versions coexist (Paraschiv, 2013). Since REST lacks a standardized approach, it is important to agree on how to handle versioning with REST, meaning how to clearly define what is versioned and how the versions are communicated (Nottingham, 2012).

In contrast, GraphQL avoids explicit versioning by continuously evolving the GraphQL schema. With new types or fields, new API capabilities can be added to already existing types, avoiding a breaking change that would require a new version (GraphQL, Schema Design, n.d.). Deprecated fields can remain available through the `@deprecated` directive (moesif, 2022). However, since existing clients rely on those schema definitions, schema changes can have significant implications (Joel, 2024).

Through Protocol Buffers gRPC achieves strong backward and forward compatibility. Developers can add new optional fields or reserve removed identifiers to prevent conflicts. This structured contract model ensures stability but requires careful version control of .proto files (Protocol Buffer Team, n.d.).

5.3.2 Debugging and Error Handling

REST supports straightforward human-readable error handling through standard HTTP status codes which can include custom error messages to provide additional details about the specific error (Albano, 2024). Logging and tracing are simple to implement through built in frameworks such as ASP.NET Core or HTTP middleware (Larkin, Gutsch, & Anderson, 2024). With Postman or browser-based API consoles, REST responses can be directly inspected, making REST easy to debug.

In GraphQL errors are structured and returned with a single JSON response that contains both data and error fields, allowing for partial success when only parts of a query fail. GraphQL provides detailed errors of different error types containing structured information such as message, location and path (Kramer, GraphQL Errors: Understanding the Basics, 2024). Tracing the origin of a specific failure, however, can be challenging. This stems from the fact that a single GraphQL request can involve multiple sub-resolvers (small functions responsible for fetching data). With built-in introspection features and tools like Apollo Studio, query performance and resolver errors can be visualized more effectively (Apollo, Apollo Server / Resolving Operations / Error Handling, n.d.; GraphQL, Learn/Introspection, n.d.).

gRPC supports structured handling of errors by using a set of predefined status codes. Additionally, advanced error details can be provided using Protobuf-defined messages (Newton-King, Error handling with gRPC, 2024; gRPC, Docs / Guides / Error handling, 2025). Logging is typically implemented through interceptors, which can capture both request metadata and responses for tracing (gRPC, Docs / Guides / Interceptors, 2024). As traditional HTTP tools cannot inspect the binary transport format used by gRPC (HTTP/2), manual inspection and debugging are more difficult (Postman Team, What is gRPC?, 2023).

5.4 Real-World Suitability assessment

This chapter assesses the practical suitability of REST, GraphQL and gRPC for various real-world scenarios based on the preceding analysis of performance, developer experience and maintainability. This way, abstract comparisons develop into clearer guidance on which API style is best for the individual use case.

5.4.1 Typical Domains and Environments

As the ubiquitous, general-purpose standard technology, REST is the primary choice for API development. This is due to the APIs simplicity, wide adoption, and strong support for standard HTTP features making REST ideal for public APIs and basic web and mobile applications (Panchal, 2024; Salma Alam-Naylor & David, 2024).

GraphQL, on the other hand, is best suited for client-driven applications such as single page web applications or mobile apps. REST's inefficiencies are directly resolved by GraphQL's ability to precisely tailor the received data to the requirements. Therefore, GraphQL is a good choice where data aggregation is frequent or client requirements change rapidly (Dilmegani, 2025; Loganathan, 2024).

gRPC excels where high performance and strict contracts are crucial. Its primary domain is internal service-to-service communication within microservice architectures. The gained efficiency by using binary Protocol Buffers and HTTP/2, make it highly suitable for real-time or other high-throughput systems (Center, gRPC: A Comprehensive Guide to Modern API Development, 2024; Synytsia, 2021).

5.4.2 Operational Considerations

Based on the APIs underlying protocol, moving to production introduces specific challenges.

REST and GraphQL are both based on standard HTTP protocol requests, which provides them with advantages associated with standard protocols. Standard tools and a mature web infrastructure allow for simpler monitoring and more straightforward integration into existing API Gateways. This way, deployment and operational barriers are lowered. Conversely, the HTTP/2 binary transport format used by gRPC can challenge existing infrastructure. Existing systems might not support gRPC features and require additional or specialized measures to ensure compatibility. Furthermore, debugging network traffic is challenging due to the binary payloads (see also Chapter 5.3.2).

5.4.3 Production Trade-offs

Each API style comes with a distinct set of trade-offs that must be considered when choosing an API style.

REST's primary trade-off is sacrificing data efficiency for universal simplicity. While it performs moderately under non-concurrent load (Chapter 5.1.1), its lack of flexibility in shaping the response causes over-fetching and larger payloads (Chapter 5.2.2). This leads to unnecessary consumption of resources and causes inefficiencies for high-data scenarios.

While GraphQL offers more flexibility, it also introduces operational and server-sided complexity. Minimizing data transfer and performing under concurrent load might justify the higher implementation effort for application that deal with highly variable client data needs.

gRPC delivers superior raw speed via its binary format and HTTP/2 transport, but this performance is traded for accessibility and ease of operation. A higher entry barrier (Protocol Buffers, steeper learning curve) and more difficult integration with standard web tools can make it less for public APIs. It is best suited as a powerful tool for internal service-to-service communication where strict contracts and maximal efficiency are required.

5.5 Comparison of results across the three API styles

This chapter compares the results discussed in the previous chapters and scores each API in the different categories and sub criteria. To provide a clear, quantifiable measure, each criterion is assigned a score from one (worst) to five (best) for the nine sub-criteria, weighted according to their importance to the overall project goal, as defined in Chapter 3.5.

The resulting analysis allows for the calculation of a final weighted score for each API, providing an evidence-based conclusion on their relative strengths in the context of the developed application. This grading chart can additionally serve as a template for engineering teams to weigh trade-offs regarding the requirements of their own projects.

Table 9 shows the final grading chart with all criteria, scores as well as the final score for each API.

Table 9. Grading chart API comparison with Final Weighted Scores.

Category	Sub-criterion	Weight	REST	GraphQL	gRPC
Performance (40%)	Latency	20%	3	4	5
	Payload Size	10%	2	4	5
	Throughput & Resources	10%	3	4	5
Developer Experience (30%)	Ease of Implementation & Implementation Time	10%	5	2	3
	Learning Curve & Tooling Support	10%	5	3	2
	Flexibility for Clients	10%	4	5	5
Maintainability (20%)	Versioning & Evolution	10%	3	5	5
	Debugging & Error Handling	10%	5	3	4
Suitability (10%)	Use-Case Fit	10%	4	4	5
Final Weighted Score		100%	3.7	3.8	4.4

The final score is calculated by summing each sub criterion with its respective weight according to the formula:

$$Weighted\ Score\ = \sum(Subcriterion\ Score\ \cdot Weight\ Percentage)$$

Latency and iteration duration for gRPC were the lowest compared to REST and GraphQL. While latency for REST and GraphQL for the small and the large query testing scenario was about even, GraphQL's slightly lower iteration durations leads to its higher overall score for latency. gRPC out-performed the other two APIs in terms of performance, securing the maximum score of five points. GraphQL's ability to prevent over-fetching and minimize data transport justifies its higher score for payload size, while REST showed, by far, the highest payload size, resulting in a lower score. gRPC shows the smallest request and response size for all three testing scenarios, once again leading to a maximum score for payload size. In terms of throughput and use of resources,

all three APIs are virtually equal for the small and large query, however GraphQL performs better and gRPC is superior under concurrent load.

Implementation time for GraphQL is by far the highest with almost double the time necessary compared to the gRPC and REST. Furthermore, especially in comparison to REST, gRPC and GraphQL have a steeper learning curve and slightly less tool support and documentation leading to comparably lower scores. On the other hand, GraphQL and gRPC have stronger type enforcement allowing for better client flexibility which leads to a higher score compared to REST with a relatively weak type enforcement.

Maintainability, specifically versioning and evolution is a weak point of REST due to the manual versioning process and risk of excess overhead or duplicated code. GraphQL, on the other hand, tries to avoid versioning and allows for easy addition of fields to the schema. Similarly, gRPC provides straight forward extensibility by manipulating the .proto file which additionally ensures backward compatibility. REST's simplicity is a major advantage when it comes to error handling, providing the developer with straightforward human readable data and errors. In contrast, for GraphQL tracing the origin of errors can be difficult and even though gRPC provides structured error handling with status codes and error details, the non-human readable binary transport format can make debugging increasingly challenging.

For this application, REST's inability to efficiently fetch nested `TaskItem` and `Comment` data makes it less suitable compared to GraphQL or gRPC. GraphQL's more efficient and precise data retrieval is a preferable match for this specific application. gRPC could also serve as a suitable API technology for this application, although its core values high performance and streaming might not be of most importance for the requirements of comparably small web applications like the one developed in this context.

The weighted comparison concludes that gRPC is the most balanced and effective solution for the task management system developed in this thesis, represented by a final weighted score of 4.4. Even though it introduces complexity during the implementation phase, its outstanding performance prevails. GraphQL secured second place with a score of 3.8, performing well in performance and maintainability but suffering from a higher implementation effort. With a score of 3.7,

REST is in third place, mainly due to the lack of performance, flexibility for clients as well as versioning and evolution of the API.

5.6 Strengths, weaknesses, and trade-offs

The preceding chapters conducted a detailed analysis of performance and functionality of all three APIs. This section summarizes the findings and highlights strengths, weaknesses , and core tradeoffs for each API. Table 10 begins this summary by detailing the specific strengths, weaknesses, and trade-offs of REST.

Table 10. REST's strengths, weaknesses, and trade-offs.

Strengths	**Weaknesses**	**Core Trade-offs**
Universal Compatibility: Mature ecosystem, full browser support, and easy integration with all standard web tools (API Gateways, proxies, caches).	Over-fetching and Under-fetching: Reply is setup of fixed data structures, leading to unnecessary data transfer and potential waste of bandwidth.	Ease of implementation and universal access are prioritized over network and client-side efficiency.
Low Entry Barrier: Quickest setup time and lowest learning curve, make it ideal for rapid prototyping and smaller applications.	Versioning Overhead: Lack of standardized versioning could lead to duplicate code and maintenance overhead.	Implementation speed and simplicity are traded for increased maintenance efforts as the API evolves.
Human-Readable Debugging: Straightforward debugging with standard HTTP status codes and JSON payloads.	High Latency under Load: Highest average latency under concurrent load potentially due to larger payload sizes.	Debugging simplicity is traded for poorer performance under high concurrency.

The analysis of REST highlights that it's fundamental strengths – universal compatibility and a low entry barrier – come at the expense of data efficiency and performance. It suggests that REST's rigid data structures create a clear limit on performance and development simplicity is prioritized over bandwidth optimization.

Table 11 shifts the focus to GraphQL, a technology designed to overcome REST's weaknesses. The table details how GraphQL optimizes for data-efficiency and client-driven data fetching, while highlighting the distinct weaknesses in increased complexity and caching challenges.

Table 11. GraphQLs strengths, weaknesses, and trade-offs.

Strengths	**Weaknesses**	**Core Trade-offs**
Client-Driven Data Retrieval: Clients can request only necessary fields, resulting in minimal payload sizes and higher data efficiency.	Increased Server-Side Complexity: Requires a dedicated schema, resolvers, and a custom data-fetching layer, increasing backend implementation complexity.	Network and data efficiency are traded for higher development complexity and a steeper learning curve.
Efficient Data Aggregation: Can replace multiple REST endpoints with a single query, reducing round trips and latency.	Caching Challenges. Requests sent via HTTP POST to a single endpoint bypass traditional HTTP caching mechanisms.	Reduced network latency is traded for the loss of simple, standardized HTTP caching.
Seamless Evolution: Breaking changes can be avoided by adding new types of fields.	Resource Intensity. Dynamic query resolution can lead to higher average CPU usage compared to REST's simpler request handling.	Cleaner long-term API evolution is traded for higher CPU cost per request.

While GraphQL successfully optimizes for minimal payloads and higher data efficiency, it introduces higher development complexity and CPU cost per request. Thus, while GraphQL effectively solves the issues of over-fetching and latency, it does so by incurring increased implementation costs and the loss of standard chaching.

Table 12 concludes the summary by detailing the attributes of gRPC. It clearly shows gRPC's main focus on maximizing efficiency and speed within internal systems and the resulting downsides from this focus.

Table 12. gRPCs strengths, weaknesses, and trade-offs.

Strengths	**Weaknesses**	**Core Trade-offs**
High Efficiency: Binary serialization and HTTP/2 transport deliver faster data exchange and lower bandwidth consumption.	Difficult Debugging: Binary transport format is not human-readable and requires specialized tooling for debugging and troubleshooting.	Increased raw performance is traded for challenging debugging and monitoring.
Strong Type Enforcement: The `.proto` files ensure strict type safety and guarantee strong forward and backward compatibility.	Steep Learning Curve: Protocol Buffers and gRPC-specific concepts can prolong onboarding time.	Reduced runtime errors due to type safety are traded for increased developer ramp-up time and complexity.
High-Frequency Data Handling: Built-in support for bi-directional streaming and fast, high-volume data exchange is optimal for internal systems.	Limited Client Accessibility: Difficult to consume directly from a web browser and poorer support by standard public API tools.	Maximum internal system efficiency is traded for reduced external accessibility and universal compatibility.

This analysis suggests that while gRPC offers supreme efficiency, its steeper learning curve and lack of native browser support might qualify it predominantly as a specialized tool rather than a general-purpose solution.

6 Discussion

6.1 Answer to Research Questions

This thesis set out to compare REST, GraphQL, and gRPC through the practical implementation of a task management application. Based on the quantitative benchmarks and qualitative analysis conducted in Chapter 5, the research questions formulated in Chapter 1.5 can now be answered.

Research Question 1: *How do REST, GraphQL, and gRPC compare in terms of performance efficiency, specifically regarding latency, payload size, and throughput under different load scenarios?*

The results clearly indicate that gRPC is the most performant architecture across all measure metrics. This is visible in the lowest latency, highest amount of requests handled per second as well as the highest throughput, especially under concurrent load. Additionally, with an average of only 78 bytes per request, gRPC showed the smallest payload size per request. Compared to REST, GraphQL demonstrated greater efficiency regarding payload size, mitigating REST's over-fetching by requesting and retrieving only the required data. In this comparison, REST proved to be the least efficient, indicated by the highest latency under load and highest payload size.

Research Question 2: *What are the trade-offs between the three API styles regarding developer experience, implementation complexity, and long-term maintainability and scalability?*

The comparison showed that performance and ease of implementation have an inverse relationship. While REST offered the best developer experience with the fewest lines of code and offered the best support in terms of tools and documentation, it suffered from a poorer maintainability in regard to versioning. GraphQL introduced the highest implementation overhead with almost double the amount of implementation time needed, but on the other hand significantly increased flexibility for clients. The strict type enforcement and use of .proto files provided gRPC with a strong backwards compatibility as well as maintainability. However, this comes at the expense of a steeper initial learning curve and increasingly difficult debugging due to the binary encoding of the data.

Research Question 3: *Based on the performance and operational trade-offs, which API architecture is most suitable for specific real-world application contexts?*

There is no universally superior API style best suited for any type of application. The most suitable API style depends strongly on the context meaning which type of application it is used for, the applications requirements, the development environment, the development team and other factors. The following Chapter 6.2 gives a more detailed description of which API type is best suited for which application.

6.2 Recommendations for practice

As explained in the preceding chapter, the comparative study of REST, GraphQL and gRPC shows that no single API technology is universally superior and best suited for all types of applications. Aligning the specific functional and performance requirements with the API's core strengths is crucial to be able to make the optimal choice of which API technology to use. Based on the practical benchmarking and subsequent analysis, the following recommendations are provided for developers having to choose an API style for their application.

REST is recommended when simplicity, universal compatibility as well as a low entry barrier are of most importance. This could include applications with straightforward CRUD operations on a single resource or access to data sets with infrequent updates. Furthermore, for APIs that need to be accessed by a broad range of clients, as is often the case for public APIs, REST is a good choice since no specialized tools or client libraries are required. Lastly, for initial development of prototypes, REST can be a good solution due to the minimal setup overhead and mature ecosystem. Conversely, REST is less suited for applications that require frequent fetching of complex and nested data or applications where payload size is critical. The inherent over-fetching of REST will lead to higher bandwidth consumption and latency, potentially causing issues when trying to meet the applications requirements.

GraphQL is recommended for more data-intensive and client-driven applications. When the application requires fetching of deeply nested data in a single request or needs to work with unreliable network connectivity, its precise query language enables a minimal payload size and high data efficiency. The schema-based evolution additionally makes GraphQL well suited for applications that

require flexibility and are exposed frequently changing frontend requirements such as mobile applications. However, developers must accept the trade-off of increased complexity imposed by the schema and resolver as well as the operational challenge related to HTTP caching.

gRPC can be the ideal choice when performance, strict typing and inter-service communication are the core requirements. If the use of efficient binary encoding of data is required or low latency and high throughput are essential, gRPC can be a good API. gRPC can also be a good fit for systems where different services are written in different languages, as Protocol Buffers generate strongly typed code for all major programming languages. Internal microservices are one of the prominent examples, gRPC's bidirectional streaming capabilities and high performance are utilized. In contrast, for simple projects or public-facing APIs gRPC might not represent the best solution, mainly due to the higher initial learning curve and lack of native browser support.

In summary, the choice among these three API technologies requires a well calculated decision which should be made before starting implementation since it can have significant impact on the applications performance as well as development and maintenance effort. REST offers accessibility at the cost of data efficiency; GraphQL offers data efficiency at the cost of implementation complexity; and gRPC offers performance at the cost of accessibility.
For the task management application benchmarked in this thesis, GraphQL offered the best balance between performance and developer complexity for a client-driven system.

6.3 Conclusions and Future Outlook

This thesis' objective was to provide a hands-on comparison of the selected APIs REST, GraphQL, and gRPC which moves beyond theoretical discussions. The implementation of a fully functional "Trello-like" task management application that includes all three APIs in the backend allowed for a successful, isolated analysis of each API's strengths and weaknesses.

In this process valuable insight was gained about how newer technologies like gRPC might offer significant performance benefits, however, are not necessarily always the perfect fit for every application. Developer experience, learning curve, maintainability and other metrics remain vital in real-word projects even if the respective API performs. As a result, frequently a compromise between flexibility and complexity or ease of use and performance is required.

As described in the limitations of this work in Chapter 1.4, this study was limited to a single type of application and did not test these APIs in a large-scale production environment. Future research could expand on this work by fully utilizing gRPC's streaming capabilities in a microservice architecture or by further investigating and potentially applying GraphQL's schema federation, where the data schema is split across multiple different servers.

This thesis concludes that developers should not by default revert to well-known and commonly used API technologies but also consider whether the more modern and advanced technologies available are the right fit for the requirements of their application. In fact, a developer should strive to have all three APIs as tools in their toolkit, to use when the specific requirements of their application calls for it.

References

Ahmad, A. (2025, August 22). *REST vs GraphQL vs gRPC*. Retrieved from Design Gurus: https://www.designgurus.io/blog/rest-graphql-grpc-system-design

Albano, J. (2024, Mai 11). *Best Practices for REST API Error Handling*. Retrieved from Baeldung: https://www.baeldung.com/rest-api-error-handling-best-practices

Apollo. (n.d.). *Apollo GraphQL*. Retrieved from Apollo Docs: https://www.apollographql.com/docs/concepts/graphql

Apollo. (n.d.). *Apollo Server / Resolving Operations / Error Handling*. Retrieved from Apollographql: https://www.apollographql.com/docs/apollo-server/data/errors

Apollo. (n.d.). *Tutorials/Lift-Off-Part-1/Schema definition language (SDL)*. Retrieved from Apollo GraphQL: https://www.apollographql.com/tutorials/lift-off-part1/03-schema-definition-language-sdl

Attique, T. (2023, November 13). *MudBlazor — Blazor Component Library — Front-end Development*. Retrieved from Medium: https://medium.com/@taha.attique/mudblazor-blazor-component-library-front-end-development-409624c49662

AWS. (n.d.). *What is a RESTful API?* Retrieved from AWS Amazon: https://aws.amazon.com/what-is/restful-api/

AWS. (n.d.). *What's the Difference Between RPC and REST?* Retrieved from AWS Amazon: https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/

Baitmangalkar, S. (2022, June 1). *Communicating Between Polyglot Services Using gRP*. Retrieved from Medium: https://medium.com/@sbaitmangalkar/communicating-between-polyglot-services-using-grpc-52da6022fecc

Bello, G. (2024, February 13). *What is Protobuf?* Retrieved from Postman Blog: https://blog.postman.com/what-is-protobuf/

Bennett, T. (2024, July 14). *REST API Principles | A Comprehensive Overview*. Retrieved from Dream Factory: https://blog.dreamfactory.com/rest-apis-an-overview-of-basic-principles

Beres, J. (2025, February 19). *Blazor Server vs. Blazor WebAssembly: Just the Facts*. Retrieved from Infragistics: https://www.infragistics.com/blogs/blazor-server-vs-blazor-webassembly/

Bhayana, D. (2025, July 23). *Resolvers in GraphQL*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/graphql/resolvers-in-graphql/

Bhayani, A. (n.d.). *Why gRPC Uses HTTP2* . Retrieved from Arpit Bhayani: https://arpitbhayani.me/blogs/grpc-http2/

Bonnin, M. (2024, September 19). *Generating type safe clients using code generation*. Retrieved from GraphQL: https://graphql.org/blog/2024-09-19-codegen/

Burk, N. (2017, Mai 1). *GraphQL SDL — Schema Definition Language*. Retrieved from Prisma: https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51

CelerData. (2024, October 29). *Understanding Data Abstraction*. Retrieved from CelerData: https://celerdata.com/glossary/a-guide-to-data-abstraction

Center, L. (2024, April 26). *gRPC: A Comprehensive Guide to Modern API Development*. Retrieved from Kong: https://konghq.com/blog/learning-center/what-is-grpc

Center, L. (2025, April 8). *What is a REST API? A Comprehensive Guide* . Retrieved from Kong: https://konghq.com/blog/learning-center/what-is-restful-api

Chandima, L. (2019, October 31). *RPC (Remote Procedure Call) summarized.* Retrieved from Medium: https://slahiruc.medium.com/rpc-remote-procedure-call-summarized-ba213a5cffb3

Darmamulia, M. A. (2024, May 16). *gRPC: a 'polyglot' Realtime Communication Services*. Retrieved from Medium: https://alifdarm.medium.com/grpc-a-polyglot-realtime-communication-services-34d941b7e37f

Dilmegani, C. (2025, September 8). *GraphQL vs. REST: Top 4 Advantages & Disadvantages*. Retrieved from AIMultiple: https://research.aimultiple.com/graphql-vs-rest/#advantages-of-graphql-over-rest

Farokhi, A. (2022, August 23). *Medium*. Retrieved from What Is gRPC and Different Types of gRPC Services: https://alirezafarokhi.medium.com/what-is-grpc-and-different-types-of-grpc-services-9b86a7267064

Fateh, D. (2025, January 15). *What is an API call?* Retrieved from contentful: https://www.contentful.com/guides/api/api-call/

Fateh, D. (2025, January 1). *What is an API endpoint?* Retrieved from contentful: https://www.contentful.com/guides/api/api-endpoint/

geeksforgeeks. (2025, September 3). *REST API Introduction*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/node-js/rest-api-introduction/

Gillis, A. S. (2024, May 13). *Remote Procedure Call (RPC)*. Retrieved from TechTarget: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC

Giroux, M.-A. (2019, June 25). *GraphQL & Caching: The Elephant in the Room*. Retrieved from Medium: https://medium.com/apollo-stack/graphql-caching-the-elephant-in-the-room-11a3df0c23ad

Goodwin, M. (2024, April 09). *What is an API (application programming interface)?* Retrieved from IBM: https://www.ibm.com/think/topics/api

GraphQL. (2025, September). *GraphQL Spec Overview*. Retrieved from GraphQL Spec : https://spec.graphql.org/September2025/#sec-Overview

GraphQL. (n.d.). *Learn / Schemas and Types*. Retrieved from GraphQL: https://graphql.org/learn/schema/

GraphQL. (n.d.). *Learn/Introspection*. Retrieved from GraphQL: https://graphql.org/learn/introspection/

GraphQL. (n.d.). *Schema Design*. Retrieved from GraphQL: https://graphql.org/learn/schema-design/

gRPC. (2024, February 29). *Docs / Guides / Interceptors*. Retrieved from gRPC: https://grpc.io/docs/guides/interceptors/

gRPC. (2025, September 22). *Docs / Guides / Error handling*. Retrieved from gRPC: https://grpc.io/docs/guides/error/

Hadzima, J. (2023, October 31). *Introduction to ASP.NET Core Blazor Server*. Retrieved from marathonus: https://marathonus.com/about/blog/introduction-to-aspnet-core-blazor-server/

Hawkins, M. (2020, June 23). *The History And Rise Of APIs*. Retrieved from Forbes: https://www.forbes.com/councils/forbestechcouncil/2020/06/23/the-history-and-rise-of-apis/

Hegde, G. (2024, October 10). *Understanding the Workflow: From .proto Files to GraphQL in a Frontend World*. Retrieved from Medium: https://medium.com/@nshganesh/understanding-the-workflow-from-proto-files-to-graphql-in-a-frontend-world-a5fb3dd6c2bd

Hygraph. (n.d.). *Learn / Schemas and Types*. Retrieved from hygraph: https://hygraph.com/learn/graphql/schemas-and-types

Hygraph. (n.d.). *What is GraphQL?* Retrieved from Hygraph: https://hygraph.com/learn/graphql

IBM. (2025, April 24). *What is REST API?* Retrieved from IBM: https://www.ibm.com/think/topics/rest-apis

Innocent, A. (2025, July 21). *How gRPC Streaming Can Make Your APIs Faster and More Reliable*. Retrieved from apidog: https://apidog.com/blog/grpc-streaming/

Innocent, A. (2025, July 19). *What is an API Server?* Retrieved from Apidog: https://apidog.com/blog/what-is-an-api-server/

Jacobson, D., Brail, G., & Woods, D. (2011). *APIs: A Strategy Guide.* O'Reilly Media.

Jadhav, S. (2024, February 8). *gRPC deep dive : Efficient network communication using HTTP/2*. Retrieved from Medium: https://jadhavsaurabh037.medium.com/grpc-deep-dive-efficient-network-communication-using-http-2-11bb97151b09

Jadhav, S. (2024, January 28). *gRPC deep dive : Introduction to gRPC*. Retrieved from Medium: https://jadhavsaurabh037.medium.com/grpc-deep-dive-introduction-to-grpc-5aa37243c132

Joel. (2024, April 21). *GraphQL pain points and how to overcome them*. Retrieved from Hygraph: https://hygraph.com/blog/graphql-pain-points

Kanjilal, J. (2024, May 10). *How to work with Dapper and SQLite in ASP.NET Core*. Retrieved from InfoWorld: https://www.infoworld.com/article/2337428/how-to-work-with-dapper-and-sqlite-in-asp-net-core.html

Kharrati, K. (2022). *Global API Management Market Size Likely to Grow at a CAGR of 26.5% By 2034.* Sandy, USA: Custom Market Insights. Retrieved from Custom Market Insights: https://www.custommarketinsights.com/press-releases/api-management-market/

Kiprono, I. (2023, August 9). *The six Guiding Principles of RESTful Architecture.* Retrieved from Medium: https://medium.com/@Ian_carson/the-six-guiding-principles-of-restful-architecture-852d707b9036

Kramer, N. (2024, March 7). *GraphQL Errors: Understanding the Basics*. Retrieved from daily.dev: https://daily.dev/blog/graphql-errors-understanding-the-basics

Kramer, N. (2024, February 24). *GraphQL Queries & Mutations: A Guide*. Retrieved from Daily Dev: https://daily.dev/blog/graphql-example-mutation-an-introductory-guide

Krantz, N. (2023, March 16). *How I discovered end-to-end type safety with GraphQL + Typescript*. Retrieved from Medium: https://medium.com/@nkrantz23/adventures-with-graphql-and-typescript-end-to-end-type-safety-is-amazing-2b2964dc7ec0

Kumar, V. (2024, April 23). *Introduction to MudBlazor — Blazor Component Library*. Retrieved from NashTech: https://blog.nashtechglobal.com/introduction-to-mudblazor-blazor-component-library/

Kundalia, R. (24, June 2025). *REST vs gRPC vs GraphQL vs WebSockets vs SOAP: A Practical Guide for Engineers*. Retrieved from dev: https://dev.to/rajkundalia/rest-vs-grpc-vs-graphql-vs-websockets-vs-soap-a-practical-guide-for-engineers-37d9?utm_source=chatgpt.com

Lane, K. (2019, October 10). *Intro to APIs: History of APIs*. Retrieved from Postman Blog: https://blog.postman.com/intro-to-apis-history-of-apis/

Larkin, K., Gutsch, J., & Anderson, R. (2024, September 18). *Learn/.NET/ASP:NET Core/Logging in .NET and ASP.NET Core*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-9.0

Le, C. (2025, February 15). *What are the benefits of using gRPC over REST in an application?* Retrieved from Medium: https://medium.com/@ltcong1411/what-are-the-benefits-using-grpc-over-rest-in-an-application-11d3ad1748f5

Loganathan, P. (2024, February 9). *API architecture showdown - Rest vs graphQL vs gRPC*. Retrieved from pradeepl: https://pradeepl.com/blog/api/rest-vs-graphql-vs-grpc/#summary

Losoviz, L. (2021, August 21). *HTTP caching in GraphQL*. Retrieved from LogRocket: https://blog.logrocket.com/http-caching-graphql/

Manager, D. (2025, November 11). *The Role of APIs in Modern Software Architecture: How They Shape Scalability, Security, and the Future of Digital Systems*. Retrieved from Deciper Zone: https://jdev.decipherzone.com/blog-detail/role-of-apis-in-modern-software-architecture

Maze, C. (2024, January 31). *ASP.NET Core Web API with EF Core Code-First Approach*. Retrieved from CodeMaze: https://code-maze.com/net-core-web-api-ef-core-code-first/

Meghna. (2024, July 11). *Blazor Server vs Blazor WebAssembly vs Blazor Web App!* Retrieved from Medium: https://medium.com/@meghnav274/blazor-server-vs-blazor-webassembly-vs-blazor-web-app-8fe8fc1cefeb

Microsoft. (2022, October 05). *Create Data Transfer Objects (DTOs)*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5

Microsoft. (2022, November 23). *Entity Framework Core - Keys*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/ef/core/modeling/keys?tabs=data-annotations

Microsoft. (2023, August 24). *Entity Framework Core Getting Started with EF Core*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/ef/core/get-started/overview/first-app?tabs=netcore-cli

Microsoft. (2024, November 12). *ASP.NET Core Razor components*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-9.0

Microsoft. (2024, January 6). *Create web APIs with ASP.NET Core*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-9.0

Microsoft. (2024, July 25). *Entity Framework Core SQLite EF Core Database Provider*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/ef/core/providers/sqlite/?tabs=dotnet-core-cli

Microsoft. (2025, August 27). *Razor Pages architecture and concepts in ASP.NET Core*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-9.0&tabs=visual-studio

Mikula, K. (2023, February 7). *The History and Evolution of APIs*. Retrieved from traefik labs: https://traefik.io/blog/the-history-and-evolution-of-apis

Miller, C. (2023, June 23). *From SOAP to REST: Tracing The History of APIs*. Retrieved from treblle: https://treblle.com/blog/from-soap-to-rest-tracing-the-history-of-apis#heading-apis-are-a-gateway-to-connectivity

Miquissene, C. J. (2020, April 28). *Understanding GraphQL and its Design Principles* . Retrieved from Dev: https://dev.to/callebdev/understanding-graphql-and-its-design-principles-fd6

moesif, A. P. (2022, March 21). *Best Practices for Versioning REST and GraphQL APIs*. Retrieved from moesif: https://www.moesif.com/blog/technical/api-design/Best-Practices-for-Versioning-REST-and-GraphQL-APIs/

Moradin, M. (2019, June 17). *Beginner's Guide to Load Testing with k6*. Retrieved from Medium: https://medium.com/swlh/beginners-guide-to-load-testing-with-k6-85ec614d2f0d

Motunrayo. (2024, May 30). *The evolution of APIs*. Retrieved from hygraph: https://hygraph.com/blog/evolution-of-apis

Mulesoft. (n.d.). *What is an API (Application Programming Interface)?* Retrieved from Mulesoft: https://www.mulesoft.com/api/what-is-an-api

Nath, S. (2023, October 3). *Request Response Model, Usages, Anatomy and Drawbacks*. Retrieved from Medium: https://medium.com/@sujoy.swe/request-response-model-usages-anatomy-and-drawbacks-42464e475cf5

Newton-King, J. (2024, July 31). *Error handling with gRPC*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/grpc/error-handling?view=aspnetcore-9.0

Newton-King, J. (2025, Mai 16). *Performance best practices with gRPC*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-9.0&utm_source=chatgpt.com

Nosowitz, D. (2024, December 19). *What is gRPC?* Retrieved from IBM: https://www.ibm.com/think/topics/grpc

Nosowitz, D., & Goodwin, M. (n.d.). *gRPC vs. REST* . Retrieved from IBM: https://www.ibm.com/think/topics/grpc-vs-rest

Nottingham, M. (2012, December 4). *Evolving HTTP APIs*. Retrieved from mnot: https://www.mnot.net/blog/2012/12/04/api-evolution

Okpala, N. (2024, February 22). *API under-fetching, and over-fetching mitigation*. Retrieved from Medium: https://medium.com/@koodu-platform/introduction-d75372443b50

Panchal, J. (2024, July 18). *What is a REST API? Steps, Benefits, Use Cases & Examples*. Retrieved from rlogical: https://www.rlogical.com/blog/what-is-rest-api-and-how-does-it-work/

Paraschiv, E. (2013, July 30). *Versioning a REST API*. Retrieved from Baeldung: https://www.baeldung.com/rest-versioning

Patel, R. (2024, October 4). *A Beginner's Guide to Entity Framework Core (EF Core)*. Retrieved from Medium: https://medium.com/@ravipatel.it/a-beginners-guide-to-entity-framework-core-ef-core-5cde48fc7f7a

Pawar, V. (2025, February 15). *Advantages and Disadvantages of REST API*. Retrieved from dev: https://dev.to/vishalpawar1010/advantages-and-disadvantages-of-rest-api-365l

Pérez, O. (2019, Mai 16). *Improve GraphQL Performance with Large Responses*. Retrieved from Travelgate: https://blog.travelgate.com/en/how-to-improve-graphql-performance?utm_source=chatgpt.com

Postman. (2022). *State of the API Report*. Retrieved from Postman: https://www.postman.com/state-of-api/2022/api-technologies/#api-technologies

Postman. (2023). *State of the API Report*. Retrieved from Postman: https://www.postman.com/state-of-api/2023/api-technologies/#api-technologies

Postman. (2024). *State of the API Report*. Retrieved from Postman: https://www.postman.com/state-of-api/2024/

Postman. (2025). *State of the API Report*. Retrieved from Postman: https://www.postman.com/state-of-api/2025/

Postman Team. (2023, November 13). *What is gRPC?* Retrieved from Postman: https://blog.postman.com/what-is-grpc/

Postman Team. (2024, January 11). *gRPC vs. GraphQL*. Retrieved from Postman Blog: https://blog.postman.com/grpc-vs-graphql/

Postman. (n.d.). *What is an API?* Retrieved from Postman: https://www.postman.com/what-is-an-api/

Protocol Buffer Team. (n.d.). *Protocol Buffers Documentation Language Guide (proto 3)*. Retrieved from protobuf: https://protobuf.dev/programming-guides/proto3

Ray, G. (2021, August 20). *Your Guide to GraphQL with TypeScript*. Retrieved from Gavin Ray: https://hasura.io/blog/your-guide-to-graphql-with-typescript#why-type-safety

Saarikoski, N. (2025). *Survey of web API definition languages.* Espoo: Aalto University. Retrieved from https://aaltodoc.aalto.fi/server/api/core/bitstreams/5689d6a8-f146-43d0-8313-21bfa5c4ee6c/content

Saleem, M. (2024, December 2). *How to Connect a SQLite Database to EF Core*. Retrieved from CodeMaze: https://code-maze.com/efcore-how-to-connect-a-sqlite-database/

Salma Alam-Naylor, & David, F. (2024, February 13). *What is a REST API?* Retrieved from contentful: https://www.contentful.com/blog/what-is-a-rest-api/

Saraiya, A. (2024, November 26). *A Deep Dive into gRPC: Service Definitions, Protobufs, and Hands-On Implementation*. Retrieved from Medium: https://medium.com/@aman-saraiya/a-deep-dive-into-grpc-service-definitions-protobufs-and-hands-on-implementation-777946d733ca

ScholarHat. (2025, September 24). *What is ASP.NET Core? Everything You need to know*. Retrieved from ScholarHat: https://www.scholarhat.com/tutorial/aspnet/introduction-to-aspnet-core

Sharma, A. (2023, July 2). *Understanding Request And Response Model: A General Overview*. Retrieved from Medium: https://medium.com/@imakashsharma135/understanding-request-and-response-model-a-general-overview-831c24d2288

Software, F. (2024, November 21). *The Role of APIs in Modern Web and Mobile Applications*. Retrieved from Medium: https://medium.com/@fulminoussoftwares/the-role-of-apis-in-modern-web-and-mobile-applications-558e2f61312a

Stemmler, K. (2021, February 23). *GraphQL Mutation vs Query – When to use a GraphQL Mutation*. Retrieved from apollographql: https://www.apollographql.com/blog/mutation-vs-query-when-to-use-graphql-mutation

Stemmler, K. (2021, February 16). *What is GraphQL? GraphQL introduction*. Retrieved from Apollo GraphQL: https://www.apollographql.com/blog/what-is-graphql-introduction#what-is-graphql

Studio, M. (2024, February 9). *Demystifying Remote Procedure Calls (RPC) for Beginners: A Comprehensive Guide*. Retrieved from Medium: https://mobterest.medium.com/demystifying-remote-procedure-calls-rpc-for-beginners-a-comprehensive-guide-7e639c92ea17

Synytsia, M. (2021, March 25). *What is gRPC: Main Concepts, Pros and Cons, Use Cases*. Retrieved from altexsoft: https://www.altexsoft.com/blog/what-is-grpc/

Team, C. (2023, December 30). *Key Elements of API Software Components and How They Work*. Retrieved from CacheFly: https://www.cachefly.com/news/key-elements-of-api-software-components-and-how-they-work/

Tech School. (2020, February 15). *HTTP/2 - The secret weapon of gRPC* . Retrieved from dev: https://dev.to/techschoolguru/http-2-the-secret-weapon-of-grpc-32dk

Vantroys, F. (2024, July 26). *The Pros and Cons of Using gRPC in Modern Software Architecture*. Retrieved from Medium: https://medium.com/@frederik_62300/the-pros-and-cons-of-using-grpc-in-modern-software-architecture-93cca0a8c8fd

Waseem, M. (2024, November 19). *What is Trello and How To Use It?* Retrieved from intellipaat: https://intellipaat.com/blog/what-is-trello/

Wilde, E. (2025, April 23). *The History of APIs: From 1947 to Modern Technologies*. Retrieved from YouTube: https://www.youtube.com/watch?v=bVlkA5W6gl8