Hochschule Merseburg
Fachbereich Wirtschafts- und
Informationswissenschaften

Bachelor Thesis

Performance Impact of the Command Query Responsibility Segregation (CQRS) Pattern in C# Web APIs

By: Jens Richter
Company: Relaxdays GmbH

Matr. No: 27300
Program of Study: Wirtschaftsinformatik

First supervisor: Prof. Dr. Sven Karol
Second supervisor: Herr Dennis Renz

Leipzig, 25. July 2024

Hochschule Merseburg
Prof. Dr. Sven Karol
Eberhard-Leibnitz-Straße 2
06217 Merseburg
Telefon + 49 3461 46 2253
E-Mail: sven.karol@hs-merseburg.de

HOME HOCHSCHULE MERSEBURG University of Applied Sciences

Aufgabenstellung für die Bachelorarbeit

von

Jens Richter

Thema:	Performance-Auswirkungen des Command Query Responsibility Segregation Patterns in C# Web APIs
Erstbetreuer:	Prof. Dr. Sven Karol
Zweitbetreuer:	Herr Dennis Renz

Aufgabenstellung

In der Entwicklung verteilter Webanwendungen sind Performance, Skalierbarkeit und Wartbarkeit wichtige Eigenschaften. Diese Eigenschaften werden durch die Implementierung von unterschiedlichen Architekturmustern (Pattern) und -prinzipien (Principles) beeinflusst. Das Command Query Responsibility Segregation (CQRS) Pattern bietet in diesem Kontext einen vielversprechenden Ansatz, indem es die Lese- (Query) von den Schreiboperationen (Command) trennt, um eine optimierte und unabhängige Skalierung beider Operationstypen zu ermöglichen.

Diese Arbeit untersucht die Auswirkungen des CQRS-Patterns auf die Performance von C# Web APIs. Hierzu wird ein komplexes, anpassbares Anwendungsszenario betrachtet, bei dem die API mit einer PostgreSQL-Datenbank interagiert. Durch einen qualitativen-empirischen Vergleich mit einer traditionellen CRUD- (Create, Read, Update, Delete) Architektur werden die potenziellen Vor- und Nachteile von CQRS hinsichtlich Systemperformance, Skalierbarkeit und Wartbarkeit gegenübergestellt. Mithilfe dieser Analyse soll aufgezeigt werden, ob und unter welchen Bedingungen die Implementierung des CQRS-Patterns in C# Web-APIs eine sinnvolle Entscheidung darstellt.

Schwerpunkte

1. Einführung in C# Web APIs anhand eines einfachen Beispiels
2. Detaillierte Betrachtung des CQRS-Patterns, seiner Komponenten und Vor-/Nachteilen
3. Diskussion der Faktoren Performance und Skalierbarkeit für Web APIs
4. Vergleich von CRUD- und CQRS-basierten Architekturen anhand eines komplexen E-Commerce Szenarios
5. Entwicklung von Testszenarien und Auswahl passender Metriken
6. Durchführung und Analyse von Performance-Tests
7. Bewertung der Eignung von CQRS für die Klasse der betrachteten Anwendungen

Prof. Dr. Sven Karol
Themenstellender Hochschullehrer

Abstract (German)

Diese Arbeit untersucht den Einfluss des Command and Query Responsibility Segregation (CQRS) (Young, 2010) Musters auf die Performance einer traditionellen Web-API, mit Schwerpunkt auf Antwortzeiten. Die Studie umfasst die Refaktorisierung einer traditionellen Web-API, die von dem E-Commerce-Unternehmen Relaxdays verwendet wird, in zwei CQRS-basierte Implementierungen: eine manuell entwickelte und die andere unter Verwendung der MediatR-Bibliothek (Bogard, n.d.). Die Web-APIs wurden unter Verwendung konsistenter Lasttest-Szenarien, die mit K6 implementiert wurden (Grafana Labs, n.d.-a), bewertet und die Ergebnisse statistisch analysiert. Die wichtigsten Erkenntnisse sind:

Performance-Analyse:

- Signifikante Unterschiede in den Antwortzeiten wurden bei folgenden Endpunkten festgestellt:
  - `GetCategoryMapping`, wo die traditionelle API signifikant besser abschnitt als beide CQRS-Implementierungen.
  - `GetChildrenOrTopLevel`, wo beide CQRS-Implementierungen die traditionelle API signifikant übertrafen.
  - `SearchCategories`, wo die traditionelle API signifikant besser abschnitt als beide CQRS-Implementierungen.
- Bei den meisten Endpunkten gab es keine wesentlichen Performance-Unterschiede zwischen der traditionellen API und den CQRS-Implementierungen.
- Die auf MediatR basierende Implementierung übertraf die manuelle CQRS-Implementierung nicht signifikant.

Mögliche Gründe für die Performance-Ergebnisse:

- Das Fehlen signifikanter Performanceverbesserungen könnte auf suboptimale CQRS Implementierungen oder die bereits effiziente traditionelle API zurückzuführen sein.
- Weitere Forschungen zur Kombination von CQRS mit Domain-Driven Design (DDD) (Evans, 2014) und Event Sourcing (ES) (Young, 2010, pp. 25–39) könnten zu signifikanteren Performanceverbesserungen führen.

Die Arbeit kommt zu dem Schluss, dass CQRS zwar architektonische Vorteile durch die Trennung von Lese- und Schreiboperationen bietet, die erwarteten Performance-Gewinne jedoch nicht konsistent beobachtet wurden. Organisationen sollten diese Erkenntnisse berücksichtigen, wenn sie sich für die Einführung von CQRS entscheiden, insbesondere in Bezug auf die potenzielle Komplexität und die Kosten für die Refaktorisierung bestehender Systeme. Zukünftige Forschungen sollten die Integration von CQRS mit DDD und ES untersuchen, andere Performance-Metriken berücksichtigen und die Optimierung von Lese-Datenbanken erforschen, um die Vorteile von CQRS in realen Anwendungen umfassend zu verstehen.

Abstract (English)

This thesis investigates the impact of the Command and Query Responsibility Segregation (CQRS) (Young, 2010) pattern on the performance of a Traditional web API, focusing on response times. The study involves refactoring a Traditional web API used by the E-Commerce company Relaxdays into two CQRS-based implementations: one manually developed and the other utilizing the MediatR library (Bogard, n.d.). The web APIs were evaluated using consistent load testing scenarios implemented with K6 (Grafana Labs, n.d.-a), and the results were analyzed statistically. The key findings include:

Performance Analysis:

- Significant differences in response times were noted in the following endpoints:
  - `GetCategoryMapping`, where the Traditional API performed significantly better than both CQRS implementations.
  - `GetChildrenOrTopLevel`, where both CQRS implementations significantly outperformed the Traditional API.
  - `SearchCategories`, where the Traditional API performed significantly better than both CQRS implementations.
- For most endpoints, there were no substantial performance differences between the Traditional API and CQRS implementations.
- The MediatR-based implementation did not significantly outperform the manual CQRS implementation.

Potential Reasons for Performance Results:

- The lack of significant performance improvement may be due to suboptimal CQRS implementations or the already efficient Traditional API.
- Further research into combining CQRS with Domain-Driven Design (DDD) (Evans, 2014) and Event Sourcing (ES) (Young, 2010, pp. 25–39) might yield more significant performance improvements.

The thesis concludes that while CQRS offers architectural benefits by separating read and write operations, the anticipated performance gains were not consistently observed. Organizations should consider these findings when deciding to adopt CQRS, especially regarding the potential complexity and cost of refactoring existing systems. Future research should explore integrating CQRS with DDD and ES, investigating other performance metrics, and optimizing read-side databases to comprehensively understand CQRS's benefits in real-world applications.

Contents

Assignment

Abstract (German)

Abstract (English)

List of Abbreviations

1 Introduction 1

2 Domain: Traditional API 4

2.1 Technology Overview 4

2.2 Domain Overview 7

2.3 Problem Space 8

2.4 Original Implementation 8

2.5 CQRS - A Possible Solution 9

3 Command and Query Responsibility Segregation (CQRS) Pattern 9

3.1 Stereotypical Architecture 9

3.2 Task-Based User Interface 10

3.3 Origins 13

3.4 Query Side 14

3.5 Command Side 15

3.6 Event Sourcing 16

3.7 CQRS and Event Sourcing 19

4 CQRS Implementation 22

4.1 Endpoint Optimizations 25

4.2 Using MediatR for CQRS Implementation 25

5 **Methodology** **27**

5.1 K6 Load Testing . . . 27

5.2 Performance Testing . . . 29

5.3 Environment . . . 45

5.4 Steps to Reproduce the Results . . . 46

5.5 Statistical Analysis . . . 47

6 **Results** **54**

6.1 Shapiro-Wilk (Normality) . . . 56

6.2 ANOVA . . . 57

6.3 Kruskal-Wallis . . . 58

6.4 Posthoc . . . 59

6.5 Box Plots . . . 60

6.5.1 GetCategoryMapping . . . 60

6.5.2 GetChildrenOrTopLevel . . . 61

6.5.3 SearchCategories . . . 62

7 **Conclusion** **64**

Listings

References

A Tables

B Figures

List of Abbreviations

API Application Programming Interface. 1–11, 17, 22, 25–30, 32, 33, 35, 36, 41, 42, 44, 45, 47, 54, 64, 65

CQRS Command and Query Responsibility Segregation. 1–3, 9, 10, 13–16, 19–22, 24, 25, 27, 41, 54, 64, 65

CQS Command and Query Separation. 13

DDD Domain-Driven Design. 2, 9–11, 15, 17, 19, 20, 64, 65

DTO Data Transfer Object. 2, 10–12, 14, 15, 20–22, 25

ES Event Sourcing. 2, 64, 65

HATEOAS Hypermedia as the Engine of Application State. 5, 6

HTTP Hypertext Transfer Protocol. 10, 28, 42

JSON JavaScript Object Notation. 5–7, 10, 29, 42

ORM Object-Relational Mapper. 7, 10, 14, 15, 18, 20, 65

RDBMS Relational Database Management System. 9, 10, 16

REST Representational State Transfer. 2, 5, 6

ROI Return on Investment. 19

SDK Software Development Kit. 1, 46

UI User Interface. 10, 11

URI Uniform Resource Identifier. 5, 6, 42

UUID Universally Unique Identifier. 12

XML eXtensible Markup Language. 5

1 Introduction

Optimizing the performance of applications is crucial for ensuring efficient user experiences. However, it is important not to optimize prematurely. As Martin Fowler notes, "Like refactoring, performance optimization does not usually change the behavior of a component (other than its speed); it only alters the internal structure. However, the purpose is different. Performance optimization often makes code harder to understand, but you need to do it to get the performance you need" (Fowler et al., 1999, p. 54).

In certain scenarios, such as in systems like heart pacemakers, where delays can have severe consequences, optimization is essential and performance is a key factor for the success of the product. But, for most systems like corporate information systems, optimization should not be prioritized early. Nonetheless, keeping optimization in mind during application design is beneficial. Fowler also suggests, "The secret to fast software, in all but hard real-time contexts, is to write tunable software first and then to tune it for sufficient speed" (Fowler et al., 1999, p. 69).

As a student and emerging software engineer with one year of experience in C#, .NET, and ASP.NET Core, I have developed an interest in design patterns that can enhance the performance and maintainability of web APIs. In my current role at the E-Commerce company Relaxdays, I face a real-world challenge: optimizing the performance of an internal C# web API. This API provides article, category, and attribute information to our content team, who rely on it to update and select categories or attributes for both new and existing articles. However, the API suffers from slow response times, creating a bottleneck that affects the team's productivity and overall efficiency.

To address this challenge, a simplified version of the company's web API, referred to as the Traditional API, will be optimized to different versions. The goal is to create a well-tested web API using the latest version of C# and .NET, as well as ensuring it is self-contained. Currently, the latest versions are C# 12 (Microsoft, 2024g) and .NET 8, both included in the .NET 8 SDK (Microsoft, 2024f). The setup involves upgrading and refactoring the Traditional API, reducing it to only the necessary models, removing internal dependencies, and minimizing network latency by running all services in Docker (Docker, 2024) containers on the same machine. Authentication and authorization are excluded to streamline the comparison.

This thesis aims to evaluate whether different implementations of the Command and Query Responsibility Segregation (CQRS) pattern can significantly improve the response times of the Traditional API, and whether CQRS can contribute to maintainability and optimization due to its separation of read and write components (Young, 2010, p. 17). The goal is to provide actionable insights into the benefits and trade-offs of adopting CQRS in similar contexts, whether implemented manually or using popular NuGet packages like MediatR (Bogard, n.d.). NuGet, the official package manager for C#/.NET, offers a variety of libraries for download and use in applications (Microsoft, 2022a). The results of this comparison will inform the company's decision-making process regarding the introduction of CQRS in the Traditional API. The focus of the study is on the performance improvements and architectural changes involved in refactoring the Traditional API to a CQRS-based implementation. This study will analyze only the base implementation changes required for the transition to CQRS, rather than detailing all aspects of the Traditional API's initial state.

Greg Young, who first described the CQRS pattern according to Martin Fowler (Fowler, 2011), wrote in his "CQRS Documents" that "CQRS and Event Sourcing become most interesting when combined together [...] [and] Domain-Driven Design has been applied" (Young, 2010, p. 50). However, to keep the scope manageable, this study only includes a theoretical explanation and does not implement Event Sourcing (ES) (Young, 2010, pp. 25–39) in combination with Domain-Driven Design (DDD) (Evans, 2014). The primary reason for this is that introducing ES, as Young suggests (Young, 2010, pp. 25–39), requires the application of DDD, which in turn necessitates a redesign of the models. Given that the models are shared across many marketplace APIs, altering them would demand extensive changes to the internal framework. Even if tests with DDD and ES demonstrated significant impacts, these results would not be transferable to the original implementation. Thus, this analysis investigates whether it is possible to utilize CQRS and achieve benefits without relying on ES and extensive refactoring to DDD. This means only simple changes or additional Data Transfer Objects (DTOs) are introduced.

The research methodology involves several key steps, starting with the Traditional API as the baseline for the CQRS implementation. The Traditional API is refactored multiple times to adopt different CQRS implementations. These versions are then tested with K6, a modern load testing tool (Grafana Labs, n.d.-a), using the same load test scenario for each endpoint. The test environment is set up using Docker (Docker, 2024) to ensure consistency and reproducibility. Data collected from the load tests underwent statistical analysis using a Python script. This provided a robust framework for evaluating performance differences between the different API implementations. The source code repository is available on https://github.com/CurvesHub/CqrsPerformanceAnalysis.

This thesis is structured into seven chapters, each addressing a specific aspect of the study:

- **Chapter 1: Introduction**

  Provides an introduction to the study, including the background, problem statement, objectives, scope and limitations, as well as an overview of the methodology.

- **Chapter 2: The Domain: Traditional API**

  Outlines the context of the domain and the technologies used, such as C#, .NET/ASP.NET Core and RESTful web APIs.

- **Chapter 3: Command and Query Responsibility Segregation Pattern**

  Discusses the Command and Query Responsibility Segregation (CQRS) pattern in detail.

- **Chapter 4: CQRS Implementations**

  Explains the initial state of the Traditional API, the steps taken to implement CQRS and the optimizations applied in each version.

- **Chapter 5: Methodology**

  Describes the research design, data collection methods, data analysis techniques and the comparative study between the Traditional API and the CQRS APIs.

- **Chapter 6: Results**

Presents the data collected from the load tests and provides a statistical analysis and comparison of the response times between the Traditional API and the CQRS APIs. Additionally interprets the results, discussing their implications for API performance.

- **Chapter 7: Conclusion**

  Summarizes the key findings of the study and suggests areas for further research based on the study's findings and limitations.

2 Domain: Traditional API

2.1 Technology Overview

The Traditional API is built using C#, .NET, and ASP.NET Core. C# is a modern, object-oriented programming language developed by Microsoft, designed for building a wide range of applications that run on the .NET Platform (Microsoft, 2024e). The .NET Platform itself is a comprehensive development platform that provides a controlled environment for developing and running applications. It offers features such as memory management, security, and robust class libraries, making it an ideal choice for developing reliable and efficient software (Microsoft, 2024c). ASP.NET Core, part of the .NET Platform, is specifically used for building dynamic web applications and services (Microsoft, 2023a).

The API is designed as a minimal web API (Microsoft, 2022b). To illustrate the difference between using Controllers versus minimal APIs in C#, a simple "Hello World" example follows. It demonstrates how the endpoint on the route `/helloWorld` can be implemented using the minimal API approach compared to the traditional Controller approach. Both implementations use top-level statements, meaning "the compiler generates a method to serve as the program entry point" without needing to define a `static void Main(string args)` method in the `Program` class (Microsoft, 2024d). The code can be written directly in the `Program.cs` file, as shown in Listing 1, where the endpoint is defined as a minimal API. The GET endpoint is mapped using the `MapGet()` method. When the API runs, it can be triggered and returns "Hello World".

```
1  // Program.cs
2  var app = WebApplication.CreateBuilder(args).Build();
3
4  app.MapGet("/helloWorld", () => Results.Ok("Hello World!"));
5
6  app.Run();
```

Listing 1: Hello World as a minimal API

As the name suggests, minimal APIs use minimal code. In this case, only three lines are required to wire up an endpoint. When using a Controller, a separate file, shown in Listing 3, is used to define the controller, which must be registered and mapped. ASP.NET Core provides two methods: `AddControllers()` registers the implementation and `MapControllers()` wires up the endpoint (Microsoft, 2024a). This leads to the following implementation in Listing 2:

```
// Program.cs
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.MapControllers();

app.Run();
```

Listing 2: Hello World using a Controller

Listing 3 shows the `HelloWorldController` class, which holds the implementation details for the endpoint.

```
// HelloWorldController.cs
using Microsoft.AspNetCore.Mvc;

namespace Example.Controllers;

[ApiController]
[Route("/helloWorld")]
public class HelloWorldController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("Hello World");
    }
}
```

Listing 3: The `HelloWorldController`

The API follows most Representational State Transfer (REST) principles, except for Hypermedia as the Engine of Application State (HATEOAS). These principles include statelessness, where each request from a client to a server must contain all the information needed to understand and process the request, ensuring that each request is independent (Fielding, 2000). Another principle is the uniform interface, which standardizes the way resources are accessed and manipulated through a consistent set of rules and conventions. The uniform interface is governed by four rules: resource identification in requests, manipulation of resources through representations, self-descriptive messages, and HATEOAS (Fielding, 2000; Fowler, 2010).

REST emphasizes the importance of resource identification through Uniform Resource Identifiers (URIs), ensuring that each resource can be uniquely accessed via a specific URI. Furthermore, REST encourages the use of representations to manipulate resources, allowing resources to be represented in different formats such as JavaScript Object Notation (JSON) or eXtensible Markup Language (XML). The final principle, HATEOAS, allows clients to dynamically navigate resources by including hypermedia

links within responses. This means that a client interacts with a RESTful service entirely through the provided hypermedia links, enabling dynamic discovery and interaction with the API (Fielding, 2000; Fowler, 2010).

However, the Traditional API does not implement HATEOAS. Including hypermedia links would make the responses larger and add unnecessary information, which is not needed since the responses are self-contained and processed by an application, not a human. If users need information about the different resources, they can use the API documentation, which follows the OPEN-API specifications (Miller et al., 2021). Every endpoint, along with its parameters and request/response models, is documented. The endpoints of the Traditional API return data in JSON format, a lightweight data interchange format that is easy to read and write for humans and easy to parse and generate for machines (Ecma International, 2017). Listing 4 shows an example response from the `/categories` endpoint.

```json
{
    "categoryNumber": 1,
    "categoryPath": "Garden",
    "germanCategoryNumber": 1,
    "germanCategoryPath": "Garten"
}
```

Listing 4: Example JSON Response from `/categories`

The Traditional API includes both POST and GET endpoints. GET endpoints enable clients to request and retrieve data from the server, such as the example in Listing 4. By specifying the URI with query parameters: `/categories?rootCategoryId=1&articleNumber=1`, the category mapping of the article in the specified category tree, indicated by the `rootCategoryId` which was sent, is returned. POST endpoints allow clients to send data to the server to create or update resources. For instance, a client might send a new category number to associate with an article if the mapping has changed or the article was never assigned to a category. This information is then stored in the database. The API provides nine endpoints for retrieving and manipulating data related to attributes and categories of articles.

- `GET /attributes`
- `PUT /attributes`
- `GET /attributes/subAttributes`
- `GET /attributes/leafAttributes`
- `GET /categories`
- `PUT /categories`
- `GET /categories/children`
- `GET /categories/search`
- `GET /rootCategories`

For managing data access, the Object-Relational Mapper (ORM) Entity Framework Core is used. Entity Framework Core simplifies data interactions by allowing developers to work with a database using C#/.NET objects (Microsoft, 2021a). ORMs are programming tools that create a bridge between object-oriented programming languages and relational databases, facilitating data manipulation and retrieval without needing to write SQL queries manually (Hibernate, n.d.). PostgreSQL, version 16.0.0, and the C#/.NET provider NpgSQL are used as the database technology (Npgsql, n.d.). PostgreSQL is a powerful and reliable relational database system known for its advanced features and performance. It ensures data integrity and supports complex queries as well as storing JSON columns, making it suitable for handling the Traditional API (PostgreSQL, 2024).

In terms of logging and monitoring, the NuGet package Serilog is utilized. Serilog is a diagnostic logging library for .NET applications that allows users to write logs to various sinks, including files, databases, and cloud services (Serilog, n.d.). In this implementation, the logs are sent to Elasticsearch using the Elasticsearch sink provided by the NuGet package Serilog.Sinks.Elasticsearch (van Oudheusden, n.d.). These logs in Elasticsearch are easily accessible and searchable through Kibana, a powerful visualization tool (Elasticsearch, n.d.). Through this setup, the API's performance and behavior can be efficiently monitored and analyzed, providing valuable insights into its operation and helping quickly identify and resolve issues.

The entire application is deployed within a Linux Docker container. Docker provides a consistent and isolated runtime environment, ensuring that the application runs the same way in development, testing, and production environments. This deployment strategy improves reliability and simplifies the deployment process, making it easier to manage and scale the application as needed (Docker, 2024).

2.2 Domain Overview

My company, Relaxdays, operates in the E-Commerce sector, specializing in the sale of over 20,000 articles, primarily in the home and garden sector. The success of its business heavily relies on maintaining a comprehensive and up-to-date product catalog (Relaxdays, 2024). Part of this process is managed by the content team, responsible for entering data for new articles and updating existing ones that could not be uploaded to the marketplace due to validation errors. Their role is vital in ensuring that the product listings are accurate, detailed, and appealing to customers, thereby driving sales and enhancing customer satisfaction.

The Traditional API serves as the data source for the content team's operations. The API offers endpoints for retrieving and manipulating data related to articles, categories, and attributes. Specifically, the API enables the content team to change or select new categories for articles and update attributes and attribute values for articles, ensuring accurate and detailed product information. This functionality is important for maintaining the quality and relevance of the product listings across different marketplaces, directly influencing market competitiveness and operational efficiency.

2.3 Problem Space

Unfortunately, the Traditional API faces significant performance issues, particularly slow response times. These delays are especially problematic during peak usage when the content team is most active. Slow response times lead to frustration and inefficiencies, as team members have to wait for, e.g., the category drop-down to load before selecting an option and waiting again. Consequently, the team's ability to quickly update and manage the product catalog decreases, impacting overall productivity and potentially resulting in delayed product listings and updates, which in turn affects the customer experience and sales.

To address these performance issues, several optimization efforts have been undertaken. The API is monitored via logs, and specific request handlers were modified when endpoints were identified as particularly slow. For instance, caches for frequently queried data that doesn't change quickly were introduced. However, these efforts have been limited in their effectiveness. The API is complex, with nine different endpoints, making it challenging to pinpoint and address all performance bottlenecks while managing daily business or new feature requirements. As a result, performance improvements have been incremental and insufficient to fully resolve the underlying issues.

2.4 Original Implementation

In the original implementation, the API is part of a suite of marketplace content APIs used by the front-end application of the content team to manage product content across different marketplaces in various languages. Each API uses a base implementation and adds marketplace-specific details or implements specific interfaces. The original API also relies heavily on several internal NuGet packages that provide a standard database schema, models, and numerous convenience functions and processes necessary for different marketplaces.

For the purposes of this thesis, several changes were made to the original implementation. All internal libraries, which would be unchangeable, were removed. The code used from internal libraries was added directly into the application, ensuring it is fully self-contained. This led to the removal of a lot of unused code, streamlining the API to only what is necessary for the thesis. The modified API now relies exclusively on freely available NuGet dependencies, ensuring that the test environment is free from external, unchangeable factors and libraries.

A significant portion of the long response times in the original implementation is caused by the internal Auth API, which is called to authenticate the user for each request. For the test scenario, this authentication mechanism has been fully removed. This was to ensure that the tests were reproducible, as varying authentication times would skew the results. Even if a constant or predefined sequence were used, it would have added a constant overhead that wouldn't have changed the test results. Additionally, in the test scenario, network latency is minimized by running the services within the same Docker engine network with a shared virtual network on the same machine. This ensures that network latency is very small and does not significantly impact the test results. These modifications ensured that the Traditional API used in this thesis was isolated from external dependencies and authentication over-

head, providing a clearer and more controlled environment for assessing the performance impact of the CQRS pattern.

2.5 CQRS - A Possible Solution

My team and I are the maintainers of the Traditional API and were looking for possible solutions. In a discussion about how to improve the response times, the Command and Query Responsibility Segregation (CQRS) design pattern was mentioned. We all had little experience with it but heard that it separates read and write operations. Separating these and then optimizing each side independently sounded like a great idea. The challenge was born: using CQRS to separate read (query) and write (command) operations, allowing for different optimization techniques for each (Young, 2010, pp. 17–23).

The separation with CQRS can help identify and eliminate database bottlenecks, ensuring that read and write queries are executed more efficiently (Fowler, 2011). By adopting CQRS, it is possible to enhance the performance of read operations, which are often the most frequent and performance-sensitive tasks for the content team. Additionally, write operations can be streamlined and optimized independently, leading to overall better performance and responsiveness (Young, 2010, p. 23). The implementation of CQRS is expected to yield several key outcomes, including improved response times for both read and write operations, resulting in a smoother and more efficient workflow for the content team. Moreover, the separated API will allow for easier optimization and scaling of individual components. Even if CQRS does not fully resolve all performance issues, it is expected to make future optimizations more effective and manageable due to its separation of concerns.

3 Command and Query Responsibility Segregation (CQRS) Pattern

3.1 Stereotypical Architecture

The "CQRS Documents" by Greg Young first discussed the CQRS pattern, as noted by Martin Fowler (Fowler, 2011). Before delving into architectures for Domain-Driven Design (DDD) (Evans, 2014) based projects, it is crucial to start by analyzing what is generally considered the standard architecture that many try to apply to projects (Young, 2010, p. 2). The Traditional API is designed in such a stereotypical architecture, which will be incrementally improved following Young's descriptions.

The architecture, as illustrated by Young, centers on a backing data storage system, typically a Relational Database Management System (RDBMS) like PostgreSQL used by the Traditional API. The important aspect of the backing store is that it represents the current state of objects in the domain, such as the mapping of categories to articles. Communication with the data storage is handled by the Application Server, which contains the business logic responsible for validating and processing requests. Abstracting the domain reveals Application Services, which provide a simple interface to the domain and underlying data while limiting coupling. In the Traditional API, these include services like `GetCategoryMappingHandler` or `GetAttributesHandler`. Users interact with the Application Server

through a Remote Facade, implemented using JSON over HTTP in the Traditional API (Young, 2010, pp. 2–3).

Clients interact with the Remote Facade in a typical DTO up/down interaction. For instance, a user might request a DTO representing a specific category, as previously shown in Listing 4. The Remote Facade loads the necessary domain objects, maps them to a DTO, and returns it to the client. The client then displays the information, allowing user interaction. After editing, the client sends the updated DTO back to the Application Server, which processes the changes, verifies them, and persists them to the data storage. The server then returns either an acknowledgment or error messages to the client. The architecture's popularity stems from its simplicity and generic nature, making it easy to teach and apply across various projects. Its generic structure allows teams to quickly adapt and become proficient in its use, reducing onboarding costs for new members. Many frameworks exist to optimize the development time for this architecture, such as ORMs for change tracking and transaction management, and automapping frameworks for DTO mappings. However, this architecture is not suitable for Domain-Driven Design (DDD) (Young, 2010, pp. 3–6).

Applying DDD within this architecture is challenging due to its focus on CRUD (Create, Read, Update, Delete) operations, which limits the richness of the Ubiquitous Language in the domain model. Ubiquitous Language refers to a common, shared language used by both developers and domain experts within a specific business domain (Evans, 2014). This often results in an anemic domain model, where domain objects have little behavior and are primarily data holders. The business logic tends to reside outside the domain, often in the client or in procedural code, undermining the principles of DDD. A significant bottleneck in this architecture is the data storage, particularly when using an RDBMS. Most RDBMS are not horizontally scalable, and vertical scaling can be prohibitively expensive. However, many systems do not require extensive scaling, so this may not be a critical issue in all cases (Young, 2010, pp. 6–7).

In summary, the DTO up/down architecture is versatile and offers simplicity for team collaboration, but it is unsuitable for DDD projects. Improving this architecture in small, rational steps while minimizing productivity costs can enhance its alignment with business needs. The subsequent sections will explore incremental improvements to this architecture, focusing on the CQRS pattern and its potential benefits (Young, 2010, p. 7).

3.2 Task-Based User Interface

The Traditional API uses task-based names for the reqeust and response DTOs used by its resources, such as `UpdateCategoryMappingRequest`. Young also describes the concept of a Task-Based User Interface (UI) and compares it with a CRUD-style User Interface. He also discusses the changes that occur within the Application Server when a more task-oriented style is applied to its API (Young, 2010, p. 9).

One of the largest problems in the stereotypical architecture is the loss of user intent. In a CRUD-based system, the client interacts by posting data-centric DTOs back and forth with the Application Server, resulting in a domain that lacks meaningful behaviors. The domain becomes a mere abstraction of the

data model, with behaviors existing only in the client, on pieces of paper, or in users' minds (Young, 2010, p. 9).

Many applications suffer from this issue, where users have documented workflows, such as "Go to screen X, edit field Y to value Z, then proceed to screen A and edit field B to value C." While this may work for low-value systems, it becomes unwieldy for complex, high-ROI systems suitable for Domain-Driven Design (DDD) (Young, 2010, p. 9).

One reason often cited for this approach is that "business logic and workflows can be changed at any time without modifying the software." However, this flexibility comes at a cost. Missed steps or inconsistent user actions can lead to unreliable system behavior and poor reporting (Young, 2010, p. 9).

To address this issue, it is beneficial to move away from the DTO up/down architecture. In the stereotypical interaction, the UI requests a DTO, such as Customer 1234, from the Application Server. The DTO is returned to the client, displayed on the screen, and interacted with by the user. After the interaction, the client sends the modified DTO back to the Application Server, which maps the data back to the domain model and saves the changes, returning a success or failure response (Young, 2010, p. 9).

However, this approach loses the user's intent because the DTO only represents the object's current state after the client's actions. By capturing user intent, the Application Server can process behaviors rather than just save data (Young, 2010, p. 9).

In a task-based interaction, the client similarly first requests a DTO from the Application Server, such as Customer 1234. The DTO is returned and displayed for user interaction. But instead of sending the modified DTO back to the server, the client sends a message indicating the user's desired action, such as "Complete a Sale," "Approve a Purchase Order," or "Submit a Loan Application." This approach allows the Application Server to understand the user's intent and execute the corresponding task (Young, 2010, pp. 10–11).

The Traditional API also uses a kind of task-based interface for its endpoints. For instance, when a client wants to get the mapped category of an article, it sends a `GetCategoryRequest` DTO to the server. The server processes the request, loads the domain objects, and returns the mapped category of the article in a `GetCategoryResponse` DTO. The client can then use the data in the response to display the category of the article.

Commands

In a task-based User Interface, the Application Server is instructed through the use of Commands. A Command is a simple object that specifies an operation and includes the necessary data to perform that operation. Commands can be thought of as Serializable Method Calls. For example, the following code represents a basic Command (Young, 2010, p. 11):

```csharp
public record DeactivateInventoryItemCommand(
    Guid InventoryItemId,
    string Comment);
```

Listing 5: Example Command `DeactivateInventoryItemCommand`

It is common practice to include the suffix "Command" in the name of the Command class. This naming convention has both advantages and disadvantages, and the decision to use it should be made carefully by the development team (Young, 2010, p. 12).

Commands are always written in the imperative tense, instructing the Application Server to perform an action. This distinction is important as it allows the server to reject the Command if necessary, unlike an Event, which indicates something that has already occurred (Young, 2010, p. 12).

An example of this linguistic distinction is the word "Purchase", which can function as both a verb in the imperative and a noun describing the result of its usage in the imperative. When creating Commands, it is essential to focus on the verb form to ensure clarity in intent. For instance, a Command should specify what to purchase and allow the domain to fill in details, rather than sending a fully detailed purchase DTO (Young, 2010, p. 12).

The example Command in Listing 5 includes two data properties: an ID representing the InventoryItem and a comment explaining why the item is being deactivated. Commands should only contain data required to process the behavior, unlike the typical architecture where the entire data object (e.g., the InventoryItem) is sent back to the server (Young, 2010, p. 12).

The ID is crucial for routing the Command to the correct object. At least one ID must be present for all Commands that update state, as Commands are intended to be directed to an object. For Create Commands, including an ID is valuable, especially in distributed systems where client-originated IDs, typically in the form of Universally Unique Identifiers (UUIDs), are beneficial (Young, 2010, p. 12).

Developers often create Commands with familiar vocabulary, such as "ChangeAddress", "CreateUser", or "DeleteClass". However, it is more effective to focus on the specific use case. For instance, "ChangeAddress" could differentiate between "Correcting an Address" and "Relocating the Customer", which is particularly relevant in contexts like a telephone company that sends yellow pages to customers' new addresses. Similarly, "CreateUser" might be better expressed as "RegisterUser", and "DeleteClass" as "DeregisterStudent". This process can lead to valuable domain insights (Young, 2010, p. 13).

Commands should be aligned with use cases, as they typically correspond to specific actions within the application. While some data in applications may only need basic CRUD operations, it is important not to confuse these with areas where use cases and intents are more complex. Although the concept of Commands is straightforward, it can be perceived as additional work for developers. If creating Commands becomes a bottleneck, it may indicate that the ideas are being applied incorrectly (Young, 2010, p. 13).

3.3 Origins

Command and Query Responsibility Segregation (CQRS) originated from Bertrand Meyer's Command and Query Separation Principle. According to Meyer, this principle states that every method should either be a command that performs an action or a query that returns data, but not both. Methods should return a value only if they don't mutate state and have no side effects. If a method mutates state, its return type must be void (Young, 2010, p. 17) (Meyer, 1997, pp. 751–752).

Martin Fowler provides an example where this principle can be bent, such as with a method that pops a stack. While Meyer advocates strict adherence to command-query separation, practical exceptions exist where modifying state and returning a value can be useful (Young, 2010, p. 17).

Initially, CQRS was considered an extension of this concept, referred to as Command and Query Separation (CQS) at a higher level. However, it evolved into a distinct pattern, maintaining the same definitions for commands and queries as Meyer used, but splitting objects into two separate entities: one for commands and one for queries (Young, 2010, p. 17).

The stereotypical architecture, as discussed in the previous chapter, typically handles both commands and queries within the same service. For example, a `CustomerService` might have methods for both actions and data retrieval (Young, 2010, pp. 17–18):

```
// ICustomerService
void MakeCustomerPreferred(CustomerId)
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)
```

Listing 6: Original Customer Service (Young, 2010, p. 18)

Applying CQRS to this service results in two separate services as shown in Listing 7 (Young, 2010, p. 19).

```
// CustomerWriteService
void MakeCustomerPreferred(CustomerId)
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

// CustomerReadService
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
```

Listing 7: Customer Service after CQRS (Young, 2010, p. 19)

This separation into read and write services enforces the notion that the Command side and the Query side have different needs, leading to various architectural benefits (Young, 2010, p. 19):

- **Consistency**: It is easier to maintain consistency for transactions on the Command side, while the Query side can often be eventually consistent (Young, 2010, p. 19). "Eventual consistency is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value." (Vogels, 2008)
- **Data Storage**: The Command side benefits from normalized data storage (near 3rd Normal Form), while the Query side benefits from denormalized storage (near 1st Normal Form) to minimize joins (Young, 2010, p. 19). Normalization in relational databases involves organizing tables to reduce redundancy and improve data integrity. 1st Normal Form (1NF) ensures each column contains only atomic values and has unique column names, eliminating repeating groups. 2nd Normal Form (2NF) is achieved when a table is in 1NF and all non-key attributes are fully dependent on the entire primary key, removing partial dependencies. 3rd Normal Form (3NF) is reached when a table is in 2NF and all non-key attributes depend solely on the primary key, eliminating transitive dependencies (Microsoft, 2024b).
- **Scalability**: The Command side generally handles fewer transactions and does not require significant scalability. In contrast, the Query side often processes a larger volume of transactions and needs to be scalable (Young, 2010, p. 19).

These differences highlight that a single model cannot optimally handle searching, reporting, and transaction processing. Therefore, CQRS recommends separate models for reads and writes, each optimized for their respective tasks (Young, 2010, pp. 19–20).

3.4 Query Side

The Query side of CQRS contains only the methods for retrieving data. In the stereotypical architecture, DTOs (Data Transfer Objects) are built by projecting from domain objects, which can be problematic due to the need for mapping between different models. DTOs are optimized to match client screens to prevent multiple server round trips, but this makes them significantly different from the domain model used for transactions (Young, 2010, p. 20).

"Common smells of the problems can be found in many domains" (Young, 2010, p. 20).

- "Large numbers of read methods on repositories often also including paging or sorting information" (Young, 2010, p. 20).
- "Getters exposing the internal state of domain objects in order to build DTOs" (Young, 2010, p. 20).
- "Use of prefetch paths on the read use cases as they require more data to be loaded by the ORM" (Young, 2010, p. 20).

- "Loading of multiple aggregate roots to build a DTO causes non-optimal querying to the data model. Alternatively aggregate boundaries can be confused because of the DTO building operations" (Young, 2010, p. 20).

The biggest challenge is optimizing queries. Since queries operate on an object model and then translate to a data model (often via an ORM), developers need deep knowledge of both the ORM and the database, leading to impedance mismatch issues (Young, 2010, p. 20).

After applying CQRS, the separation between command and query paths becomes explicit, allowing for a new approach to project DTOs without using the domain model. This introduces the concept of a "Thin Read Layer," which directly reads from the database and projects into DTOs. This layer can be implemented using a range of tools, from handwritten SQL queries and mapping code to a full ORM. The choice depends on the team's comfort level and specific needs, with a middle-ground solution often being most practical (Young, 2010, pp. 20–22).

The Thin Read Layer does not need to be isolated from the database and can be tied to a specific database vendor. Using stored procedures for reading can also be beneficial, depending on the team's preferences and system requirements. While maintaining this layer can be tedious, it avoids impedance mismatch by directly connecting to the data model, making queries easier to optimize. Developers working on the Query side only need to understand the data model, not the domain model or ORM tools. "The separation of the Thin Read Layer and the bypassing of the domain for reads allows also for the specialization of the domain" (Young, 2010, pp. 20–22).

3.5 Command Side

The Command side in CQRS remains similar to the "Stereotypical Architecture", with the primary difference being its behavioral rather than data-centric contract. This shift is necessary for implementing Domain-Driven Design (DDD) and involves separating read operations from the domain. Although Young emphasizes using DDD, for this study, the domain models are not refactored to rich domain models because this would not easily be possible for the real application, other than the read DTOs which can simply be separated from the domain model (Young, 2010, p. 22).

In the "Stereotypical Architecture", the domain handled both commands and queries, leading to the previously mentioned issues. By separating the read layer, the domain focuses solely on processing commands, eliminating these issues. Domain objects no longer need to expose internal states and repositories have fewer query methods (primarily GetById) (Young, 2010, p. 23).

This separation can generally be achieved with low or no additional cost compared to the original architecture. In many cases, it can lower costs, as query optimization is simpler in a thin read layer than within the domain model. The architecture also reduces conceptual overhead, as querying is separated, potentially lowering costs further. In the worst case, costs remain equal, with responsibilities merely redistributed. It is even possible for the read side to still use the domain if necessary (Young, 2010, p. 23).

By applying CQRS, reads and writes are separated, raising the question of whether they should share the same data model. Treating them as two integrated systems allows each data model to be optimized for its specific task. For example, the read side can be modeled in 1NF, while the transactional model can be in 3NF (Microsoft, 2024b) (Young, 2010, p. 23).

The choice of integration model is crucial, as translation and synchronization between models can be costly. Introducing events, a well-known integration pattern, offers the best mechanism for model synchronization, maintaining consistency efficiently (Young, 2010, p. 23). The next sections will discuss what an Event Store is and how Event Sourcing and CQRS fit together. However, to keep the scope of the thesis manageable and the results more comparable to the real application, the Event Store and Event Sourcing are not implemented in this study.

3.6 Event Sourcing

Most production systems today rely on storing the current state to process transactions. Historically, before the widespread adoption of RDBMS as the architectural center, many high-performance, mission-critical, and secure systems did not store the current state. Interestingly, even RDBMSs themselves often manage data without focusing solely on the current state. To capture what happened before the current state was in place, domain events can be used (Young, 2010, p. 25).

A domain event represents something that has occurred in the past. All events should be named using past-tense verbs, such as `CategorySaved`, `CargoShipped`, or `OrderFulfilled`. This naming convention is crucial as it integrates the event into the Ubiquitous Language (Evans, 2014), making the side effects of actions explicit and well-defined. In code, an event is a simple data-holding structure, similar to a Command but with different intent. Commands ask the system to perform an operation, whereas events record actions that have already occurred (Young, 2010, pp. 25–26).

Martin Fowler provides an example of a domain event: "I go to Babur's for a meal on Tuesday, and pay by credit card. This might be modeled as an event, whose event type is `MakePurchase`, whose subject is my credit card, and whose occurred date is Tuesday" (Fowler, 2005). Fowler suggests that recording inputs as domain events helps organize processing logic and maintain an audit log. However, Fowler's example may be better suited as a Command (`MakePurchase`) rather than an event. The event should be in the past tense, such as `PurchaseMade`, reflecting an action that has already happened (Young, 2010, pp. 26–27).

The distinction between Commands and Events is crucial. Commands are in the imperative form, requesting an action (e.g., `PlaceSale`), while events are in the past tense, recording completed actions (e.g., `SaleCompleted`). This separation clarifies the intent and ensures that commands can be rejected if necessary, while events represent immutable historical facts. It also addresses several issues. For example, if a client requests an action that the domain cannot fulfill due to insufficient information, it is natural for the domain to reject the command (`PlaceSale`). It is less logical for the domain to refute a past event (`SaleCompleted`) as if it could undo history. This linguistic clarity aids both developers and domain experts, reducing the cognitive load and potential for errors (Young, 2010, p. 27).

By separating Commands and Events, developers gain a clearer understanding of context, improving the maintainability and scalability of the system. This explicit language distinction also helps new team members ramp up more quickly and ensures that anyone consuming the API can understand its behavior more intuitively (Young, 2010, p. 27).

Events as a Storage Mechanism

When considering object storage, most people think structurally. For example, a "sale" might be stored as a "Sale" object with "Line Items" and "Shipping Information." However, this is not the only approach. Another method involves using events to record changes over time, which offers different architectural benefits (Young, 2010, pp. 27–28).

Consider the creation of a small `Order` object for a web-based sale system. Typically, developers envision a structural model: an `Order` has `Line Items` and `Shipping Information`. This viewpoint focuses on the order's structure and its parts. However, data can also be viewed as a series of transactions or deltas, representing changes from one point to the next (Young, 2010, pp. 27–28).

In many mature business models, such as accounting, transactions (deltas) are recorded. Each transaction is applied to the last known value to calculate the current state. This method creates a verifiable audit log, allowing the state of an account to be reconstructed from the beginning of time. This approach has several advantages. It allows viewing the state at any point in time, which is useful for debugging and ensuring compliance with business rules. For example, if an account should not drop below zero, one can easily examine the state before the invalid transaction to understand what went wrong (Young, 2010, pp. 27–28).

In Domain-Driven Design (DDD), every domain is naturally transaction-based, focusing on behaviors and use cases. Returning to the `Order` example, the order can be represented as a series of transactions or events, such as `ItemAdded`, `ItemRemoved`, and `OrderShipped`. Replaying these events returns the object to its last known state. This approach decouples the representation of the current state in the domain from the persistence mechanism. It allows the domain model to evolve independently of the storage structure (Young, 2010, pp. 28–30).

It is crucial to use past-tense verbs for events, such as `ItemAdded` or `OrderShipped`, to reflect that these actions have already occurred. Using imperative verbs, like "Add Item" or "Create Order," could lead to issues. For instance, reconstituting an object should not trigger behaviors like reserving inventory, which might have changed over time. This distinction between commands and events is essential. Commands, which request actions, are in the imperative tense. Events, which record actions that have occurred, are in the past tense. This separation ensures clarity and prevents the issues that arise when reconstituting objects or handling logic changes (Young, 2010, pp. 29–30).

In event-sourced systems, deleting information does not mean removing past events. Instead, deletions are modeled as new transactions, called Reversal Transactions, which negate the effect of previous events. For instance, if two pairs of socks were added to a shopping basket and later removed, the final state reflects their removal, but the history shows they were once added. This approach leaves a trail, providing a clear audit log and maintaining data integrity. Architecturally, this append-only model

simplifies distribution and reduces the need for data locks, enhancing system scalability. Storing events as an append-only model is easier to scale compared to traditional relational models. This model simplifies optimization, as it involves a single append-only structure. The system benefits from reduced complexity and improved performance due to the straightforward nature of appending events (Young, 2010, pp. 31–32).

Loading objects in relational systems typically involves multiple queries to build an aggregate, often mitigated by techniques like Lazy Loading (Microsoft, 2021b). However, Lazy Loading adds complexity and can slow down operations. In event-sourced systems, all events for an aggregate are loaded and replayed, requiring only a single query. This method avoids the need for Lazy Loading, simplifying the development process and reducing the learning curve for developers. A solution for handling the concerns about a large number of events to load effectively is discussed in the next section. A Rolling Snapshot is a denormalization of the current state of an aggregate at a specific point in time, representing the state after all events up to that point have been replayed. This technique helps avoid the inefficiency of loading all events from the beginning of time. Instead, by using snapshots, only the events from the snapshot point forward need to be replayed to reconstruct the aggregate state. Those snapshots can be taken asynchronously, used to control the number of events processed during loading, and be tuned to optimize performance significantly (Young, 2010, pp. 32–35).

Using events as a storage mechanism addresses the impedance mismatch between object-oriented domain models and relational databases. Scott Ambler explains that this mismatch arises because the object-oriented paradigm is based on software engineering principles, while the relational paradigm is based on mathematical principles. This fundamental difference leads to a non-ideal combination, requiring developers to understand both paradigms and make intelligent trade-offs. ORMs (ORM) help minimize impedance mismatch but still impose a cost. “To succeed using objects and relational databases together you need to understand both paradigms, and their differences, and then make intelligent trade-offs based on that knowledge” (Ambler). The subtle differences between them include (Young, 2010, pp. 36–37):

- **Declarative vs. Imperative Interfaces**: Relational systems use data as interfaces, while object-oriented systems use behavior.
- **Structure vs. Behavior**: Object-oriented methods focus on program structure, while relational systems focus on runtime behavior and efficiency.
- **Set vs. Graph Relationships**: Relational systems use set theory, while object-oriented systems use graph theory.

These differences complicate development and require significant knowledge to manage effectively. In contrast, event-sourced systems do not suffer from this mismatch. Events are a domain concept, and replaying events to reach a state is inherently aligned with the domain model. This alignment reduces the need for multiple representations of the model and simplifies the developer’s understanding of the system. By defining everything in domain terms, event-sourced systems lower the cognitive load on developers, eliminate the need for complex ORM tools, and provide a more coherent and manageable architecture (Young, 2010, pp. 36–37).

The value of an Event Log is closely tied to the contexts where Domain-Driven Design (DDD) is beneficial. DDD is complex and costly but offers high Return on Investment (ROI) in domains where the business derives competitive advantage. Similarly, using an Event Log provides high ROI in such contexts but may not be cost-effective elsewhere. Storing only the current state of data limits the types of questions that can be answered. For example, in stock market orders, knowing the current state is often sufficient to determine liquidity. However, this approach focuses on "what" the data is at a given point in time, suitable for queries like inventory levels or sales figures. Increasingly, businesses are interested in "how" something came to be, which is central to Business Intelligence. Understanding behaviors and patterns, such as the correlation between actions and purchases, requires tracking the history of changes, not just the current state (Young, 2010, pp. 37–38).

Consider a development team at an online retailer exploring a domain expert's hypothesis: there might be a correlation between items added and then removed from carts and later purchases. A team using a current state mechanism would need to start tracking this behavior from the point the feature is implemented, limiting historical analysis to recent data. In contrast, a team using Event Sourcing can backpopulate historical data by replaying events. This allows them to generate reports with comprehensive historical insights, providing immediate and deep analysis from the system's inception. By storing events, they can retrospectively interpret old data in new ways, offering significant business insights. The ability to retrospectively analyze data is invaluable in dynamic business environments. Businesses often find new ways to look at data, and having an event log allows for flexible and comprehensive analysis. The potential competitive advantage gained from this capability can be substantial, making the initial investment worthwhile (Young, 2010, pp. 38–39).

Predicting how a business will need to look at data in the future is challenging. While concepts like YAGNI (You Ain't Gonna Need It) (Wikipedia, 2023) argue against over-engineering, the unpredictable nature of business demands flexibility in data analysis. The costs associated with modeling every behavior and storing every event are justified by the potential high ROI (Return on Investment) when the business derives its competitive advantage from this data. In summary, while Event Sourcing involves higher initial costs, the ability to analyze historical data in versatile ways provides a strategic advantage, especially in domains where understanding behavior and patterns are important (Young, 2010, pp. 38–39).

3.7 CQRS and Event Sourcing

CQRS and Event Sourcing become particularly powerful when combined, especially within systems that apply Domain-Driven Design (DDD). Their symbiotic relationship allows Event Sourcing to serve as the data storage mechanism for the domain, addressing several challenges and optimizing the architecture (Young, 2010, p. 50).

One significant issue with Event Sourcing alone is the inability to query the system directly for specific data, such as "Give me all users whose first names are 'Greg'", due to the lack of a current state representation. With CQRS, the only query required within the domain is GetById, which Event Sourcing supports efficiently. Event Sourcing is important for building non-trivial CQRS-based systems. Integrating different models (relational models for read and write) is costly, especially when maintaining

synchronization between them. Event Sourcing mitigates this by making the event model the persistence model on the Write side, eliminating the need for conversions between models and thus reducing development costs (Young, 2010, p. 50).

The client-side development costs are identical for both architectures since the client interacts with DTOs and produces Commands similarly in both models. Query implementation costs are also comparable. In the traditional architecture, queries are built from the domain model, whereas, in CQRS, they are constructed by the Thin Read Layer projecting directly to DTOs. As previously discussed, the Thin Read Layer is either equally or less expensive to implement (Young, 2010, pp. 51–52).

The primary cost difference lies in the domain model and persistence. In traditional architectures, an ORM manages the persistence of the domain model within a relational database, introducing an Impedance Mismatch between the domain model and the storage mechanism. This mismatch is costly in terms of productivity and the required developer knowledge. In contrast, CQRS and Event Sourcing eliminate the Impedance Mismatch on the Write side. The domain produces and stores events directly, which is all the domain model needs to understand. However, there is still an impedance mismatch on the read model, where Event Handlers update the read model based on events. This mismatch is smaller and easier to bridge since the event model represents actions rather than structures (Young, 2010, pp. 51–52).

While the event-based model may involve slightly higher initial costs due to the definition of events, it offers numerous benefits, such as reduced impedance mismatch and the ability to leverage event-driven insights, which offset these costs. Consequently, the CQRS and Event Sourcing model is often less expensive overall. In summary, the combination of CQRS and Event Sourcing provides a robust and cost-effective architecture that simplifies domain and data model integration, enhances query efficiency, and offers significant long-term benefits for systems leveraging DDD (Young, 2010, p. 52).

So far, the discussion has focused on isolated systems. However, organizations often rely on integrated systems of systems. In stereotypical architectures, integration is not supported out-of-the-box, except perhaps integration through the database, which is usually discouraged. Integration tends to be an afterthought, requiring custom solutions. Organizations often build services over their model to facilitate integration, which can result in substantial additional work. This becomes particularly challenging when the product is delivered to many customers. Development teams must provide hooks for all customers, accommodating various integration needs. This often leads to a large, complex codebase, especially for systems installed at numerous different clients, each with unique requirements. Although billing clients for custom integration can be profitable, it results in a cumbersome and inefficient software model (Young, 2010, pp. 52–53).

In contrast, CQRS and Event Sourcing consider integration from the very first use case. The Read side must integrate and represent the actions occurring on the Write side, making integration a fundamental part of the architecture. This approach ensures that the integration model is "production ready" and continuously tested throughout development. The event-based integration model is inherently complete, as all system behaviors generate events. If the system can perform an action, it is automatically integrated. While it may be necessary to limit the publishing of certain events, this is a matter of configuration rather than additional coding. Furthermore, the event-based model naturally supports a push model, which has advantages over the pull model used in traditional architectures. Implementing a push model in

a traditional system requires significant effort to track and synchronize events with the system's data model. In contrast, CQRS and Event Sourcing inherently support this approach, simplifying integration and ensuring consistency (Young, 2010, pp. 52–53).

The work habits in traditional and CQRS/Event Sourcing architectures differ significantly, particularly in terms of parallelization. Traditional architectures often follow one of four methodologies (Young, 2010, p. 53):

- **Data Centric**: Start with the database and work outward.
- **Client Centric**: Start with the client and work inward.
- **Façade/Contract First**: Start with the façade, then work back to the data model, and finally implement the client.
- **Domain Centric**: Start with the domain model, work outward to the client, then implement the data model.

These methodologies typically work in vertical slices, where the same developers handle a feature through all steps. In contrast, CQRS and Event Sourcing architectures can be viewed as three distinct, decoupled areas: the client, the domain, and the read model. The client consumes DTOs and produces Commands, the domain consumes Commands and produces events, and the read model consumes events and produces DTOs. This decoupled nature offers significant advantages for team dynamics (Young, 2010, pp. 53–54).

Traditional vertical slicing (Wikipedia, 2024g) works well for small teams of five to eight developers, as communication remains manageable. However, scaling up the number of developers increases the complexity of coordination and the potential for conflicts. With CQRS and Event Sourcing, work can be divided into three concurrent vertical slices: the client, the domain, and the read model. This division allows for better scaling of the development team, as each area is isolated and changes cause less conflict. It is feasible to nearly triple the team size without increasing communication overhead, as developers focus on smaller, decoupled pieces. This approach is especially beneficial when time to market is critical, as it can significantly reduce the overall calendar time required to complete a project by enabling more parallel work streams (Young, 2010, pp. 54–55).

In a development team, there can be differences among developers in terms of (Young, 2010, p. 55):

- Technical proficiency
- Knowledge of the business domain
- Cost
- Soft skills

The decoupling inherent in CQRS and Event Sourcing supports specialization. For example, working on the domain model requires developers with high technical proficiency, extensive business knowledge,

and excellent soft skills to interact with domain experts. Conversely, generating DTOs in the read model is straightforward and does not demand the same level of expertise (Young, 2010, p. 55).

Outsourcing parts of a project can be cost-effective, especially in high-cost locales. However, it often fails due to communication challenges, such as time differences, cultural differences, and language barriers. But with CQRS the read model is ideal for outsourcing because its contracts and specifications are clear and require minimal business knowledge. Mid-range technical proficiency is sufficient. In contrast, the domain model should not be outsourced. Developers working on the domain model need constant communication with domain experts and benefit from having initial domain knowledge. These developers are best kept locally within the team and should be highly valued. By outsourcing the read model, a company can save significant capital, which can be reinvested in more critical areas of the system, optimizing the overall cost and quality (Young, 2010, pp. 55–56).

Working in vertical slices often means that the most valuable developers spend only 20-30% of their time on the domain. In the CQRS and Event Sourcing architecture, developers focused on the domain spend over 80% of their time interacting with domain experts, focusing solely on use cases and understanding Commands and Events. This specialization enhances the modeling process and the development of a highly descriptive Ubiquitous Language with domain experts. While the separation of tasks in CQRS and Event Sourcing offers many benefits, it is not always necessary. Smaller teams can still work in vertical slices, balancing risk management and overall system understanding. The real advantage of CQRS and Event Sourcing is the flexibility to optimize team attributes by splitting the work into distinct vertical slices, each tailored to specific needs (Young, 2010, p. 56).

4 CQRS Implementation

This section describes the steps taken to implement the Command and Query Responsibility Segregation (CQRS) pattern in the Traditional API. It includes a description of the necessary code changes, the use of MediatR (Bogard, n.d.), and the optimizations made to improve performance. The complete source code repository is available on https://github.com/CurvesHub/CqrsPerformanceAnalysis.

The transition from the Traditional API implementation to the CQRS API implementation in the `CqrsPerformanceAnalysis` solution involved a series of methodical steps aimed at restructuring the codebase to adopt the CQRS pattern.

Project Creation: A new project named `CqrsApi` was created within the `CqrsPerformanceAnalysis` solution. The entire codebase from the existing `TraditionalApi` project was copied into the new CQRS API project to serve as a baseline for further modifications.

Functional Testing: To ensure both implementations functioned correctly, a dedicated functional test project (`CqrsApi.Tests`) was created. The functional tests were copied and run against each API to verify their correctness before proceeding with further changes.

Splitting Contexts and Services: The database context, services, and repositories were split into distinct read and write components. This separation was necessary to align with the CQRS principle of segregating read and write operations (Young, 2010). The read context was configured with

`NoTracking` to optimize read operations. "Tracking behavior controls if Entity Framework Core keeps information about an entity instance in its change tracker" (Microsoft, 2023c).

CqrsReadDbContext Protection: A safeguard was added to the `CqrsReadDbContext` to throw an exception if any of the `SaveChanges()` methods were called, ensuring that write operations could not be performed in the read context. The changes are shown in Listing 8:

```
public override int SaveChanges()
{
    ThrowOnSaveChanges();
    return 0;
}

public override int SaveChanges(bool acceptAllChangesOnSuccess)
{
    ThrowOnSaveChanges();
    return 0;
}

public override Task<int> SaveChangesAsync(
    bool acceptAllChangesOnSuccess,
    CancellationToken cancellationToken = default)
{
    ThrowOnSaveChanges();
    return Task.FromResult(0);
}

public override Task<int> SaveChangesAsync(
    CancellationToken cancellationToken = default)
{
    ThrowOnSaveChanges();
    return Task.FromResult(0);
}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSnakeCaseNamingConvention();
    optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}

private static void ThrowOnSaveChanges()
{
    throw new InvalidOperationException(
        "SaveChanges cannot be called on read context.");
}
```

Listing 8: `CqrsReadDbContext` protection

File Structure Reorganization: The file structure within the solution was reorganized to clearly distinguish between query and command operations. The original structure is shown in Listing 9:

```
- UseCases
    - Attributes
        - Common
        - GetAttributes
        - GetLeafAttributes
        - GetSubAttributes
        - UpdateAttributeValues
    - Categories
        - Common
        - GetCategoryMapping
        - GetChildrenOrTopLevel
        - SearchCategories
        - UpdateCategoryMapping
    - RootCategories
        - Common
        - GetRootCategories
```

Listing 9: Traditional API structure

It was refactored to the CQRS structure as shown in Listing 10:

```
- UseCases
    - Attributes
        - Commands
            - UpdateAttributeValues
        - Common
        - Queries
            - GetAttributes
            - GetLeafAttributes
            - GetSubAttributes
    - Categories
        - Commands
            - UpdateCategoryMapping
        - Common
        - Queries
            - GetCategoryMapping
            - GetChildrenOrTopLevel
            - SearchCategories
    - RootCategories
        - Common
        - Queries
            - GetRootCategories
```

Listing 10: CQRS API structure

Correspondingly, request suffixes were renamed to either `Command` or `Query`. For example, the class `GetCategoryMappingRequest` became `GetCategoryMappingQuery`.

4.1 Endpoint Optimizations

For all endpoints, the repositories are inlined within handlers and services, where the methods have only one usage. This simplifies optimizations in the CQRS API. To ensure a fair comparison, the inlining is done for both the Traditional API and the CQRS API.

Optimization Per Endpoint:

- **GetRootCategories**: The handler is removed, and the repository is accessed directly within the endpoint for both the Traditional API and the CQRS API to maintain fairness. This endpoint cannot be further optimized due to the necessity of using a cached repository to minimize database queries.
- **SearchCategories**: The repository is inlined in the handler. Additionally, for the CQRS API, a `SearchCategoryDto` is created to project only the necessary data and avoid loading the category entity.
- **GetChildCategoriesOrTopLevel**: Similar to `SearchCategories`, the repository is inlined in the handler, and projections are used to directly fill the response DTO, avoiding the loading of the entire category entity.
- **GetCategoryMapping**: The repository is inlined, and projections are used to directly populate the response DTO, bypassing the domain article entity.
- **UpdateCategoryMapping**: The repository is inlined in the handler, with no further optimization possible due to the need to save changes using Entity Framework Core (Microsoft, 2021a) and domain entities.
- **Attribute Use Cases**: For the attribute use cases, the repository methods are also inlined within the handlers. Further optimization is limited since slim DTOs are already in use or the attribute entity data is essential for filling the response DTO.

Consistency in Domain Models: The domain/entity models and the database schema remained unchanged, ensuring all APIs utilized the same database structure. The only exception is the class holding the `Category` entity, where the `IsSelected` property is removed and loaded directly into the response DTOs. However, this property was previously ignored by Entity Framework Core and not mapped to the database.

4.2 Using MediatR for CQRS Implementation

Usage of MediatR: MediatR was chosen as the second implementation of the CQRS pattern due to its popularity in the C# community, with over 211.8 million total downloads on nuget.org (Bogard, 2024). It "supports request/response, commands, queries, notifications and events, synchronous and asynchronous with intelligent dispatching via C# generic variance" (Bogard, n.d.), which can be leveraged in further research. The CQRS MediatR implementation can be found on a separate branch called `main-rebased-mediatr`.

MediatR Integration: The integration of MediatR is straightforward, focusing on command and query handling. All minimal API endpoints are decoupled from the handlers (Bogard, n.d.).

Handler Replacement: The handler calls are replaced with the `ISender` interface, utilizing the `Send()` method. Listing 11 shows the changes to the `SearchCategoriesEndpoint`. To use MediatR with generic types, specific query classes inherit from the `BaseQuery`. These include `GetAttributesQuery` and `GetCategoryMappingQuery`, addressing the limitation of the `BaseQuery` object in implementing different response types for different handlers.

```
private static async Task<IResult> SearchCategoriesAsync(
    [AsParameters] SearchCategoriesQuery query,
    // Before: SearchCategoriesQueryHandler queryHandler
    [FromServices] ISender sender,
    [FromServices] HttpProblemDetailsService problemDetailsService)
{
    // var result = await queryHandler.SearchCategoriesAsync(query);
    var result = await sender.Send(query);

    return result.Match(
        categories => Results.Ok(ToResponse(categories)),
        problemDetailsService.LogErrorsAndReturnProblem);
}
```

Listing 11: `SearchCategoriesEndpoint` with MediatR

Interface Implementation: To wire up the MediatR endpoints, each query/command has to implement the `IRequest` or `IRequest<out TResponse>` interface, and corresponding handlers implement the `IRequestHandler<in TRequest>` or `IRequestHandler<in TRequest, TResponse>` interface. Listing 12 and Listing 13 illustrate the usage of the interfaces on the `SearchCategoriesQuery` and `SearchCategoriesQueryHandler`.

```
public record SearchCategoriesQuery(
    int RootCategoryId,
    string ArticleNumber,
    long? CategoryNumber = null,
    string? SearchTerm = null)
    : BaseQuery(RootCategoryId, ArticleNumber),
      IRequest<ErrorOr<IEnumerable<SearchCategoryDto>>>;
```

Listing 12: `SearchCategoriesQuery` with MediatR

```
public class SearchCategoriesQueryHandler(CqrsReadDbContext _dbContext)
    : IRequestHandler<
        SearchCategoriesQuery,
        ErrorOr<IEnumerable<SearchCategoryDto>>>
{
    public async Task<ErrorOr<IEnumerable<SearchCategoryDto>>> Handle(
        SearchCategoriesQuery query, CancellationToken cancellationToken)
    {
        // ...
    }
}
```

Listing 13: `SearchCategoriesQueryHandler` with MediatR

Dependency Injection: All registered handlers are removed from dependency injection, and the following call is used to register all handlers through their implemented interfaces:

```
// In class DependencyInjection in the CqrsApi
services.AddMediatR(cfg => cfg.RegisterServicesFromAssembly(
    typeof(DependencyInjection).Assembly));
```

Listing 14: Dependency Injection: MediatR handlers

5 Methodology

This section outlines the systematic approach used to evaluate the performance impact of different implementations of the Command and Query Responsibility Segregation (CQRS) pattern in the Traditional API. It includes details of the load testing process with K6 (Grafana Labs, n.d.-a), the research design, and the statistical methods used for analyzing the results.

5.1 K6 Load Testing

K6 is an open-source load testing tool that helps developers assess the performance and reliability of their applications. It allows for simulating numerous virtual users making requests to a system, which helps in identifying potential bottlenecks and performance issues. K6 uses a JavaScript-based scripting language, making it straightforward for developers to write and maintain test scripts while providing detailed metrics for performance analysis and optimization (Grafana Labs, n.d.-a, n.d.-c).

Each of the nine endpoints tested has its own script, all sharing the same base structure but varying in the specific requests they send. These scripts are located in the `PerformanceTest` project directory at `Assets/K6Tests`. Listing 15 displays the `K6-GetAttributes` script:

```
import http from 'k6/http';
import { sleep, randomSeed } from 'k6';

export const options = {
    // Test duration: 2min
    // Purpose: Test the performance when the number of users increases,
                plateau and decreases
    stages: [
        { duration: '30s', target: 15 },    // Ramp-up from 1 to 15 users
        { duration: '1m', target: 15 },     // Stay at 15 users for 1 minute
        { duration: '30s', target: 0 },     // Ramp-down from 15 to 0 users
    ]
};

// Env seed for determinism
const seed = __ENV.MY_SEED;
randomSeed(seed);

// Generate a random integer from 1 to max (inclusive)
function randomInt(max) {
    return Math.floor(Math.random() * max) + 1;
}

const baseUrl =
    http://host.docker.internal:${__ENV.API_PORT_TO_USE}/attributes;

function randomUrl() {
    return `${baseUrl}?rootCategoryId=1&articleNumber=${randomInt(10000)}`;
}

export default function () {
    http.get(randomUrl());
    sleep(1); // Sleep for 1 second
}
```

Listing 15: `K6-GetAttributes` script

The script performs the following actions:

- Imports necessary modules from K6, including `http` for making HTTP requests and `sleep` and `randomSeed` for controlling the flow of the test.
- Sets up the test options, specifying the stages of the test.
- Sets a seed for the random number generator to ensure deterministic behavior.
- Defines a function `randomInt(max)` to generate a random integer from 1 to `max`.
- Specifies the base URL for the API being tested, using the `__ENV.API_PORT_TO_USE` environment variable for the port number.
- Defines a function `randomUrl()` to generate a URL for the API with a random article number.

- In the main function called during the test, it makes a GET request to the randomly generated URL and then sleeps for 1 second, simulating a user making a request and then waiting before making the next request.

The `options` variable is consistently defined across all scripts to ensure comparability. It holds the configuration for virtual user stages, simulating a typical workday with increasing user load over time, peaking, and then decreasing towards the end. The stages are configured as follows:

- Ramp-up from 1 to 15 users over 30 seconds
- Stay at 15 users for 1 minute
- Ramp-down from 15 to 0 users over 30 seconds

Additionally, the `__ENV.MY_SEED` is used in each script to inject the seed for generating a sequence of random numbers from one to ten thousand.

At the end of a load test with K6, a detailed summary of the test results is generated. This summary includes key metrics such as response times, data transfer rates, and the number of successful requests. K6 also allows users to export this summary using the `-out` flag, enabling the results to be saved in various formats, including JSON (Grafana Labs, n.d.-b). All summary results are archived as JSON files in the corresponding `Assets/K6Tests/[UseCase]/archive` directory. Additionally, the `metrics.jsonl` and `report.html` files are saved for further research.

5.2 Performance Testing

To run the K6 tests automatically and achieve consistent results, a custom C# web API called `PerformanceTests` was created. This API manages the execution of K6 tests within Docker containers (Docker, 2024) and collects the results, which are then processed and stored in a PostgreSQL database (PostgreSQL, 2024) for further analysis.

The `PerformanceTests` API provides a GET endpoint for each use case endpoint, enabling the execution of automated K6 tests and data collection:

- `K6Tests/attributes/getAttributes`
- `K6Tests/attributes/getLeafAttributes`
- `K6Tests/attributes/getSubAttributes`
- `K6Tests/attributes/updateAttributeValues`
- `K6Tests/categories/getCategoryMapping`
- `K6Tests/categories/getChildrenOrTopLevel`
- `K6Tests/categories/searchCategories`

- `K6Tests/categories/updateCategoryMapping`
- `K6Tests/rootCategories/getRootCategories`

In addition to the primary test endpoints, the API includes several maintenance endpoints that assist with database setup and data management, although they are not used in the automated tests:

- `database/createMainDb` - Recreates the main database for local API testing.
- `database/createPerformanceDb` - Manages the performance database where test results are stored.
- `manualTesting/addExampleData` - Inserts example data, such as articles and categories, into the main database.
- `manualTesting/processResults` - Processes the results from K6 tests.
- `K6Tests/allOfOneApi` - Executes all K6 tests for a specified API.

The most critical endpoint is `GET K6Tests/finalTestRunWithThreeSeeds`, which executes a comprehensive set of tests across nine endpoints using three different implementations and three different seeds, totaling 81 tests (9 endpoints × 3 implementations × 3 seeds). The implementation details of this endpoint are provided in Listing 16:

```
private static async Task<IResult> StartAllFinalK6TestAsync(
    ILogger logger,
    IServiceProvider serviceProvider,
    CancellationToken cancellationToken,
    int testsPerApi = 1)
{
    const bool checkElastic = true, withWarmUp = true, saveMinimalResults
        = true;
    string[] seeds = ["hardcoded_seed", "another_seed", "third_seed"];

    // Add 3 APIs with 9 endpoint tests for 3 seeds
    // -> 3*9*3 = 81 Tests a 2min = 162min = 2h 42min
    List<TestInformation> testInfos = [];
    foreach (var seed in seeds)
    {
        testInfos.AddRange(GetAllTestInfosPerEndpointForApi(
            AvailableApiNames.TraditionalApi, checkElastic,
            withWarmUp, saveMinimalResults, seed));
        testInfos.AddRange(GetAllTestInfosPerEndpointForApi(
            AvailableApiNames.CqrsApi, checkElastic, withWarmUp,
            saveMinimalResults, seed));
        testInfos.AddRange(GetAllTestInfosPerEndpointForApi(
            AvailableApiNames.CqrsApiMediatr, checkElastic, withWarmUp,
            saveMinimalResults, seed));
    }

    if (testsPerApi > 1)
    {
        // Copy the testInfos for each additional test
        var additionalTestInfos = new List<TestInformation>(testInfos);
        for (int i = 1; i < testsPerApi; i++)
        {
            testInfos.AddRange(additionalTestInfos);
        }
    }

    LogEstimatedRuntime(logger, testInfos);

    int testCount = 1;
    foreach (var testInfo in testInfos)
    {
        await using var scope = serviceProvider.CreateAsyncScope();
        var handler = scope.ServiceProvider.GetRequiredService<K6TestHandler>();
        await handler.StartK6TestAndProcessResultsAsync(
            testInfo, cancellationToken);
        logger.Information(
            "Handled test {CurrentTestCount} of {TotalTestCount}",
            testCount++, testInfos.Count);
    }

    return Results.Ok();
}
```

Listing 16: `StartAllFinalK6TestAsync`

The `StartAllFinalK6TestAsync` method initiates and processes a series of K6 tests. It accepts four parameters: an `ILogger` instance for logging, an `IServiceProvider` to access necessary services, a `CancellationToken` for task cancellation, and an optional `testsPerApi` parameter (defaulting to 1) that specifies the number of tests per API.

Initially, the method defines several constants and an array of seed strings for the K6 tests. It then generates a list of `TestInformation` objects for each combination of seed and API, using the `GetAllTestInfosPerEndpointForApi` method to gather test information. If `testsPerApi` is greater than 1, the list of `testInfos` is duplicated accordingly. The estimated runtime of the tests is logged using the `LogEstimatedRuntime` method.

Subsequently, the method iterates over the `testInfos` list, executing and processing each K6 test. For each test, a new scope is created from the `serviceProvider`, a `K6TestHandler` is retrieved, and the `StartK6TestAndProcessResultsAsync` method is called. The current test count is logged after processing each test.

The method concludes by returning an `Ok` result, indicating the successful initiation and processing of all tests.

Listing 17 presents the helper method `GetAllTestInfosPerEndpointForApi`. This method is designed to generate a list of `TestInformation` objects for various API endpoints, with each `TestInformation` object representing a specific test to be conducted.

```
private static List<TestInformation> GetAllTestInfosPerEndpointForApi(
    AvailableApiNames apiToUse,
    bool checkElastic,
    bool withWarmUp,
    bool saveMinimalResults,
    string seed)
{
    return
    [
        TestInformation.CreateInfoForGetAttributes(
            apiToUse,
            checkElastic,
            withWarmUp,
            saveMinimalResults,
            seed),
        // ...
    ];
}
```

Listing 17: `GetAllTestInfosPerEndpointForApi` method

This method takes five parameters:

- `apiToUse`: Specifies the API to be tested (type `AvailableApiNames`).
- `checkElastic`: A boolean indicating whether to check Elastic.

- `withWarmUp`: A boolean indicating whether to include a warm-up phase in the test.
- `saveMinimalResults`: A boolean indicating whether to save minimal results.
- `seed`: A string representing the seed to be used in the test.

Illustrated in Listing 17, one method for each endpoint, like `CreateInfoForGetAttributes` and `CreateInfoForGetLeafAttributes`, generates `TestInformation` objects for testing the respective endpoints, such as `GetAttributes` and `GetLeafAttributes` of the API. The parameters passed to these methods configure the tests appropriately.

Listing 18 displays the properties of the `TestInformation` class, which encapsulates information related to testing an API endpoint. This class is a record type in C#, providing built-in functionality for data encapsulation (Microsoft, 2023b).

```
public record TestInformation
{
    public string EndpointName { get; }
    public string TestDirectoryName { get; }
    public AvailableApiNames ApiToUse { get; }
    public string EndpointRoute { get; }
    public object? WarmUpRequest { get; }
    public bool CheckElastic { get; }
    public bool WithWarmUp { get; }
    public bool SaveMinimalResults { get; }
    public string Seed { get; }

    // ...
}
```

Listing 18: `TestInformation`

The `TestInformation` class has several properties to store test information, such as the endpoint name (`EndpointName`), the directory for scripts and results (`TestDirectoryName`), the API to use (`ApiToUse`), the endpoint route (`EndpointRoute`), and more. The class includes a private constructor that initializes these properties, taking parameters corresponding to each property.

```
private TestInformation(
    string endpointName,
    string testDirectoryName,
    string endpointRoute,
    object? warmUpRequest,
    AvailableApiNames apiToUse,
    bool checkElastic,
    bool withWarmUp,
    bool saveMinimalResults,
    string seed)
{
    EndpointName = endpointName;
    TestDirectoryName = testDirectoryName;
    ApiToUse = apiToUse;
    EndpointRoute = endpointRoute;
    WarmUpRequest = warmUpRequest;
    CheckElastic = checkElastic;
    WithWarmUp = withWarmUp;
    SaveMinimalResults = saveMinimalResults;
    Seed = seed;
}
```

Listing 19: `TestInformation` constructor

The class also includes several static methods that create instances of `TestInformation` for different endpoints. Each method takes parameters similar to the constructor and uses them to create a new `TestInformation` instance with the appropriate values.

For example, the `CreateInfoForGetAttributes` method is defined as follows:

```
public static TestInformation CreateInfoForGetAttributes(
    AvailableApiNames apiToUse,
    bool checkElastic,
    bool withWarmUp,
    bool saveMinimalResults,
    string seed)
{
    return new TestInformation(
        endpointName: "GetAttributes",
        testDirectoryName: "Attributes",
        endpointRoute: EndpointRoutes.Attributes.GET_ATTRIBUTES,
        warmUpRequest: new BaseRequest(RootCategoryId: 1, ArticleNumber: "1"),
        apiToUse: apiToUse,
        checkElastic: checkElastic,
        withWarmUp: withWarmUp,
        saveMinimalResults: saveMinimalResults,
        seed: seed);
}
```

Listing 20: `CreateInfoForGetAttributes` method in `TestInformation`

The K6TestHandler

The `K6TestHandler` class is responsible for managing the automated execution of K6 tests for specific API endpoints. This class handles the preparation, execution, and result processing of the tests, ensuring that all necessary containers and data setups are properly managed.

The class constructor takes several dependencies, as shown in Listing 21:

- `ILogger _logger`: Used for logging information throughout the test process.
- `PerformanceDbContext _performanceDbContext`: Provides access to the performance database context.
- `ContainerProvider _containerProvider`: Manages the lifecycle of Docker containers used during testing.
- `TestDataGenerator _testDataGenerator`: Sets up example data required for tests.
- `ApiClient _apiClient`: Interacts with the API being tested.
- `ResultProcessor _resultProcessor`: Processes the results generated from the K6 tests.

```
public class K6TestHandler(
    ILogger _logger,
    PerformanceDbContext _performanceDbContext,
    ContainerProvider _containerProvider,
    TestDataGenerator _testDataGenerator,
    ApiClient _apiClient,
    ResultProcessor _resultProcessor)
{
    // ...
}
```

Listing 21: `K6TestHandler` constructor

The `StartK6TestAndProcessResultsAsync` method displayed step-by-step in the following listings holds the core function of the `K6TestHandler` class. It orchestrates the entire process of running a K6 test, from preparation to result processing.

First, the method logs the initiation of the test for a specific endpoint and ensures that all required containers are running:

```
public async Task StartK6TestAndProcessResultsAsync(
        TestInformation testInformation,
        CancellationToken cancellationToken = default)
{
    _logger.Information("Preparing the K6 test for: {EndpointName}",
        testInformation.EndpointName);

    try
    {
        await ThrowIfRequiredContainersAreNotRunningAsync(
            testInformation.CheckElastic, cancellationToken);

        _logger.Information(
            "The required messuring/result containers are running. "
            + "Starting the api and database containers");
// ...
```

Listing 22: `StartK6TestAndProcessResultsAsync` method - Part 1

Next, it starts the necessary API and database containers and sets up the database and example data using `_testDataGenerator`. Additionally, it waits for the API to be ready for testing:

```
// ...
await _containerProvider.StartApiContainerAsync(
    testInformation.ApiToUse, cancellationToken);
await _containerProvider.StartDbContainerAsync(cancellationToken);

_logger.Information("Creating the database and setting up the example data");

await _testDataGenerator.SetupExampleDataAsync(
    cancellationToken: cancellationToken);

await _apiClient.WaitForApiToBeReady(
    testInformation.ApiToUse, cancellationToken);
// ...
```

Listing 23: `StartK6TestAndProcessResultsAsync` method - Part 2

If a warm-up phase is required, it sends warm-up requests to the API (for the final tests, 10 warm-up requests were used):

```
// ...
if (testInformation.WithWarmUp)
{
    await _apiClient.SendWarmupRequestsAsync(
        testInformation.ApiToUse,
        testInformation.EndpointRoute,
        testInformation.WarmUpRequest,
        cancellationToken);

    _logger.Information("{ApiType} API is warmed up",
        testInformation.ApiToUse);
}
// ...
```

Listing 24: `StartK6TestAndProcessResultsAsync` method - Part 3

It then executes the K6 test and waits for its completion, capturing the exit code and logs:

```
// ...
_logger.Information("Starting the k6 test for: {EndpointName} "
    + "and waiting for it to finish", testInformation.EndpointName);

var (exitCode, logs) = await _containerProvider
    .StartK6TestAndGetK6ExitCodeAndLogsAsync(
        testInformation.TestDirectoryName,
        testInformation.EndpointName,
        testInformation.ApiToUse,
        testInformation.Seed,
        cancellationToken);

_logger.Information("The k6 test for: {EndpointName} is stopped "
    + "with exit code {ExitCode}. Processing the results",
    testInformation.EndpointName, exitCode);
// ...
```

Listing 25: `StartK6TestAndProcessResultsAsync` method - Part 4

Finally, it processes the test results and cleans up the containers after the test is completed or if it failed with an exception:

```
// ...
        await _resultProcessor.ProcessResultsAsync(
            testInformation, logs, cancellationToken);
    }
    finally
    {
        await _containerProvider.CleanupAsync();
        _logger.Information("Finished K6 test for: {EndpointName}. " +
            "The containers are stopped, disposed and being deleted",
            testInformation.EndpointName);
    }
}
```

Listing 26: `StartK6TestAndProcessResultsAsync` method - Part 5

The `ThrowIfRequiredContainersAreNotRunningAsync` method verifies that all necessary containers are running before starting the test. This method helps ensure the testing environment is correctly set up and is shown in Listing 27. It performs the following checks:

- Ensures the performance database is accessible.
- Optionally checks if the ElasticSearch container is running.
- Throws an `InvalidOperationException` if any required container is not running.

```
private async Task ThrowIfRequiredContainersAreNotRunningAsync(
    bool checkElastic,
    CancellationToken cancellationToken = default)
{
    if (!await _performanceDbContext.Database
            .CanConnectAsync(cancellationToken))
    {
        throw new InvalidOperationException(
            "The performance db is not running.");
    }

    if (checkElastic)
    {
        var client = new HttpClient();

        var elasticResponse = await client.GetAsync(
            "http://localhost:9200", cancellationToken);

        if (!elasticResponse.IsSuccessStatusCode)
        {
            throw new InvalidOperationException(
                "Elastic search is not running.");
        }
    }
}
```

Listing 27: `ThrowIfRequiredContainersAreNotRunningAsync` method

Further implementation details of the services used by the `K6TestHandler` can be viewed in the source code repository at https://github.com/CurvesHub/CqrsPerformanceAnalysis/tree/main-rebased/tests/PerformanceTests/Common/Services. The functionality and the operations they provide for the `K6TestHandler` are explained in the following subsections.

The TestDataGenerator

The `TestDataGenerator` class is designed to generate and manage example data for the K6 test execution. The main functionality is encapsulated in the public method `SetupExampleDataAsync`, which performs a series of tasks to set up the example data asynchronously. It accepts the optional parameter `customDataCount` to specify the number of articles to create and `cancellationToken` to handle task cancellation. The default number of articles is 10,000 if not provided.

The method `SetupExampleDataAsync` performs the following operations:

- **Database Preparation**
  - Calls `CreateDatabaseAsync` to ensure the database is created and ready for use.
- **Data Restoration**

- Checks if a local dump file with the required data count exists. If it does, the method restores the database from this dump file using the `RestoreDump` method and logs the activity, then returns.

- **Data Creation**
  - Logs the start of data creation.
  - Retrieves the German root category from the database, which serves as the root for all categories and attributes.
  - Creates a specified number of articles using the `ArticleFactory`.

- **Categories and Attributes Assignment**
  - For each article, a category tree with a depth of 3 is created.
  - Categories are assigned to articles in a cyclic manner.
  - Variants of each article are created and added to the list of articles.
  - Attributes are created for each category and linked to articles and their variants.

- **Database Insertion**
  - Adds the created articles, categories, and attributes to the database context.
  - Saves changes to the database and clears the change tracker to reset the state of the context.

- **Dump Creation**
  - Creates a database dump file to be used for future restorations.

- **Logging**
  - Logs various stages of the process, including the completion of data creation, the number of changes saved, and the total elapsed time.

It utilizes a couple of helper methods to facilitate the above operations, such as `CreateDump`, `ExecuteShellCommand`, `CreateCategories`, `CreateAttributes`, and others. Their purposes are explained below.

- `CreateDatabaseAsync`

  Ensures the database is deleted and recreated to start with a clean slate.

- `CreateDump`

  Creates a database dump file using a shell command executed within a Docker container.

- `RestoreDump`

  Restores the database from an existing dump file, again using a shell command in Docker.

- `ExecuteShellCommand`

  Executes shell commands and handles their output and errors.

- `CreateCategories`

  Generates a tree of categories with a depth of three levels, linking them appropriately.

- `CreateAttributes`

  Generates attributes for the given categories and assigns them to articles and their variants.

- `GetLocalDumpFilePathWithName` and `GetDockerDumpFilePathWithName`

  Generate file paths for local and Docker dump files respectively.

In summary, the `TestDataGenerator` class provides a comprehensive mechanism for setting up example data in a database environment, supporting both creation and restoration processes. This allows for consistent and repeatable data setup, which is essential for performance testing. The implementation details of this service can be viewed in the source code repository at https://github.com/CurvesHub/CqrsPerformanceAnalysis/blob/main-rebased/tests/PerformanceTests/Common/Services/TestDataGenerator.cs.

The ApiClient

The `ApiClient` class is designed to provide an interface for interacting with either the Traditional, CQRS, or CQRS Mediatr API. The class offers two public methods: `WaitForApiToBeReady` and `SendWarmupRequestsAsync`.

The method `WaitForApiToBeReady` ensures that the specified API is ready to accept requests. It performs a series of readiness checks to confirm that the API is operational before proceeding with any further operations. The method `SendWarmupRequestsAsync` sends a series of warm-up requests to the specified API to prepare it for subsequent operations.

The operations of the `WaitForApiToBeReady` method are as follows:

- **Logging**:

  Logs the beginning of the waiting process for the API to be ready.

- **Delay and Initialization**:

  Waits for 3 seconds to allow initial setup and initializes a new `HttpClient` with the base address set to either the Traditional, CQRS, or CQRS Mediatr API, based on the `apiToUse` parameter.

- **Readiness Check Loop**:

  Attempts to send a GET request to the root categories endpoint. Logs the attempt and checks if the response indicates the API is ready. If the API is not ready, catches `HttpRequestException`, increments an exception counter, and logs the failure. Repeats the process for a maximum of 10 attempts, each separated by a 1-second delay. If the API is still not ready after 10 attempts, logs a warning and throws an `InvalidOperationException`.

The `SendWarmupRequestsAsync` method sends a series of warm-up requests to the specified API. It performs the following steps:

- **Logging**:

  Logs the beginning of the warm-up process.

- **API Address Setup**:

  Sets the base address of the `_client` to the appropriate API based on the `apiToUse` parameter.

- **Warm-up Requests Loop**:

  Sends 10 warm-up requests to the specified route. Uses the `SendRequestBasedOnType` method to determine the type of request to send based on the `request` parameter. Logs each attempt and waits for 250 milliseconds between requests.

It utilizes a couple of helper methods to facilitate the above operations. These methods are private and are used internally by the `SendWarmupRequestsAsync` or `WaitForApiToBeReady` method. Their purposes are explained below.

The `SendRequestBasedOnType` method sends a request to the API based on the type of the `request` parameter. It supports different request types by switching on the type of `request` and performing the corresponding Hypertext Transfer Protocol (HTTP) method (GET, PUT) with the appropriate URI or JSON body. If the response indicates failure, it throws an `InvalidOperationException`.

These other methods help construct the appropriate URIs for various request types:

- `CreateRequestUriForSearchCategories`
- `CreateRequestUriForGetChildrenOrTopLevel`
- `CreateRequestUriForGetSubAttributes`
- `CreateRequestUriForGetLeafAttributes`
- `CreateRequestUri`: Modifies the query parameters of the URI based on the request type.
- `CreateUriBuilderFromBaseRequest`: Constructs a `UriBuilder` from the base request and route.

The `ApiClient` class provides a structured way to interact with one of the APIs, ensuring it is ready before making requests and sending warm-up requests to prepare the API for subsequent operations. The comprehensive logging ensures that each step is documented for troubleshooting and monitoring purposes. The implementation details of this service can be viewed in the source code repository at https://github.com/CurvesHub/CqrsPerformanceAnalysis/blob/main-rebased/tests/PerformanceTests/Common/Services/ApiClient.cs.

The ResultProcessor

The `ResultProcessor` class provides functionality to process the results of K6 tests. The class includes a public method `ProcessResultsAsync` and several private helper methods to manage the processing of test results.

The method `ProcessResultsAsync` performs the following steps:

- **Logging Start**: Logs the start of the result processing.
- **Directory Retrieval**: Retrieves all directory paths named `results` within the `Assets/K6Tests` directory.
- **Processing Each Result Directory**: For each result directory, processes the files and constructs `TestRun` objects.
- **Database Insertion**: Adds the processed test run data to the `PerformanceDbContext` and saves the changes.
- **Logging Completion**: Logs the completion of result processing, including the number of changes saved and the counts of test runs and data points.

The class includes several private helper methods to support result processing. These methods are private and are used internally by the `ProcessResultsAsync` method. Their purposes are explained below:

- `GetTestName`:

  Extracts the test name from the file paths by finding the file containing `Summary` in its name.
- `GetCreationDateTimeWithoutSeconds`:

  Gets the creation date and time of a file, excluding seconds.
- `SetMetricAndDataTypes`:

  Sets the metric and data types for data points based on the provided metric definitions.
- `ProcessResultDirectoryAsync`:

  Asynchronously processes the files in a result directory.
- `GroupedFiles`:

  Groups files by their creation time without seconds to handle multiple test runs for the same test.
- `ConstructTestRun`:

  Reads the summary content, data points, and HTML report from the grouped files and constructs a `TestRun` object.

- `ExtractDataPointsAsync`:

  Extracts data points from a file, deserializes lines containing metric definitions and metric points, filters out non-essential metric points if minimal results are requested, and sets the metric and data types for the data points.

- `ArchiveFiles`:

  Archives the processed files by moving them to an archive directory.

In summary, the `ResultProcessor` class provides a structured approach to processing K6 test results, managing the data extraction, transformation, and storage operations. The comprehensive logging facilitates troubleshooting and monitoring. The implementation details of this service can be viewed in the source code repository at https://github.com/CurvesHub/CqrsPerformanceAnalysis/blob/main-rebased/tests/PerformanceTests/Common/Services/ResultProcessor.cs.

The ContainerProvider

The `ContainerProvider` class provides functionality for managing Docker containers used in automated tests. It defines methods to start, stop, and clean up containers, specifically for database, API, and K6 load testing.

The class includes constants for resource limits and maintains references to the Docker container objects:

- `LIMIT_TO_ONE_CPU`: Limits the container to 1 CPU.
- `LIMIT_TO_ONE_GB_OF_MEMORY`: Limits the container to 1 GB of memory.
- `_dbContainer`: Reference to the PostgreSQL database container.
- `_apiContainer`: Reference to the API container.
- `_k6Container`: Reference to the K6 load testing container.

The class provides the following public methods:

- `CleanupAsync`: Stops and removes all containers.
- `StartApiContainerAsync`: Configures and builds the API container using the private method `BuildApiContainer` and starts it.
- `StartDbContainerAsync`: Configures and builds the database container using the private method `BuildDbContainer` and starts it.
- `StartK6TestAndGetK6ExitCodeAndLogsAsync`: Configures and builds the K6 container using the private method `BuildK6Container`, starts the K6 load test, waits for it to finish, and retrieves the exit code and logs.

The `ContainerProvider` class uses the `Testcontainers` NuGet package to manage the Docker containers (Hofmeister et al., n.d.). The configuration and build of the database, API, and K6 containers are handled by private helper methods:

- `BuildDbContainer`:

  Builds the PostgreSQL database container with the specified configurations. Sets the image, network, name, database, username, password, and port binding. Additionally, it sets the resource limits to 1 CPU and 1 GB of memory and configures the mount point for database dumps.

- `BuildApiContainer`:

  Builds the API container based on the specified API type. Sets the image, network, name, and port binding based on the API type. Additionally, it sets the environment to `Production` and resource limits to 1 CPU and 1 GB of memory.

- `BuildK6Container`:

  Builds the K6 container for load testing. Sets the image, network, name, and bind mounts for scripts and results. Configures the K6 command and environment variables and sets resource limits to 2 CPUs and 2 GB of memory.

In summary, the `ContainerProvider` class provides a structured approach to managing Docker containers for automated testing. It supports starting, stopping, and cleaning up containers for database, API, and K6 load testing. The helper methods facilitate the configuration and management of containers. The implementation details of this service can be viewed in the source code repository at https://github.com/CurvesHub/CqrsPerformanceAnalysis/blob/main-rebased/tests/PerformanceTests/Common/Services/ContainerProvider.cs.

5.3 Environment

A consistent test environment is crucial for reproducibility in performance testing. The hardware and software specifications used in our testing are as follows:

Hardware:

- Machine: Laptop
- Processor (CPU): 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2.80 GHz
- Memory (RAM): 32.0 GB (31.7 GB usable)

Software:

- Operating System: Windows 10 Pro 22H2 64-bit OS, x64-based processor
- Docker Desktop: 4.29.0 (145265)

- Docker Engine: 26.0.0
- Performance Database: PostgreSQL 16.2
- Elastic Container: Elasticsearch 8.13.0
- .NET Software Development Kit (SDK): 8.0.206
- K6: 0.51.0
- Python: 3.9-slim

5.4 Steps to Reproduce the Results

Before any tests were run, and only once for all tests, the Docker images for the `TraditionalApi`, `CqrsApi`, and `CqrsApiMediatr` were built. Additionally, the `PerformanceTestApi` was published using the .NET SDK with the final test stages into a separate directory.

To ensure the reproducibility of the results, the following steps were followed for each execution of the `GET K6Tests/finalTestRunWithThreeSeeds` endpoint:

1. Plugged the laptop into power to prevent any power-saving interruptions.
2. Restarted the laptop to ensure a clean state.
3. Set the laptop to the highest performance power mode.
4. Deactivated WLAN to avoid network interference.
5. Closed the applications Slack and PowerToys.
6. Opened Docker Desktop and started the performance database and Elasticsearch containers.
7. Opened two PowerShell terminals for executing the commands.

Then, the following commands were executed in the first terminal:

- Navigated to the `PerformanceTests` directory.
- Started the performance test project using:

  `dotnet PerformanceTests.dll -urls="http://localhost:5090".`

In the second terminal, the following commands were executed:

- Sent a request to ensure the application was running:

  `curl 'http://localhost:5090'`

- Upon successful response, started the final test run with the parameter `testsPerApi` set to 3:

  `curl 'http://localhost:5090/K6Tests/finalTestRunWithThreeSeeds?testsPerApi=3'`

Each execution of the `/finalTestRunWithThreeSeeds` endpoint runs 81 tests (3 seeds $\times$ 3 APIs $\times$ 9 endpoints), each running for 2 minutes. The total test duration for the ten planned runs is 27 hours. As part of the study, a total of 810 tests were run, and their results were collected. Table 1 below outlines the total duration for varying numbers of test runs:

Number of Runs	Duration (hours)
1 run	2.7 hours
2 runs	5.4 hours
3 runs	8.1 hours
4 runs	10.8 hours
5 runs	13.5 hours
6 runs	16.2 hours
7 runs	18.9 hours
8 runs	21.6 hours
9 runs	24.3 hours
10 runs	27 hours

Table 1: Test Duration for Different Number of Runs

5.5 Statistical Analysis

To analyze the processed test run data, a systematic approach is employed, ensuring a comprehensive understanding of the performance results. The analysis process involves querying specific parts of the result summary data for each test using an SQL script, saving the queried data as a CSV file, and then performing a detailed statistical analysis using a Python script. All relevant files are located in the `PerformanceTests/Assets/ResultAnalyzer` directory. The steps are as follows:

1. Query the relevant parts of the result summary data of each test with the SQL script: `ResultAnalyzer/dataQuery.sql`
2. Save the data as a CSV file: `ResultAnalyzer/mounted/k6_performance_data.csv`
3. Analyze the data with a Python script: `ResultAnalyzer/mounted/ResultAnalyzer.py`

The script utilizes several statistical methods to provide a thorough analysis of the data:

1. **Descriptive Statistics**: Descriptive statistics involve summarizing and organizing data to highlight its essential features. This includes measures of central tendency (mean, median, mode) and measures of variability (range, variance, standard deviation). Descriptive statistics provide a snapshot of the data set, describing its general characteristics without making inferences about

a larger population. They can also involve graphical representations such as histograms and box plots, aiding in data visualization and interpretation (Wikipedia, 2024c).

2. **Box Plots**: Box plots, or box-and-whisker plots, visually represent data through their quartiles, showing the distribution, central tendency, and variability. They include a box that spans the interquartile range (IQR) from the first quartile (Q1) to the third quartile (Q3), with a line at the median (Q2). Whiskers extend from the box to the smallest and largest values within 1.5 times the IQR, while outliers are plotted as individual points (Wikipedia, 2024b).

3. **Normality Tests (Shapiro-Wilk)**: The Shapiro-Wilk test is a statistical test used to assess the normality of a data set. It evaluates the null hypothesis that a sample comes from a normally distributed population. The test calculates a statistic, W, which measures how well the data follow a normal distribution. A low W value indicates a departure from normality. If the p-value is less than the chosen alpha level, the null hypothesis is rejected, suggesting that the data are not normally distributed (Wikipedia, 2024f).

4. **ANOVA (Analysis of Variance)**: ANOVA (Analysis of Variance) is a statistical method developed by Ronald Fisher to analyze differences among group means by partitioning observed variance into components attributable to different sources of variation. It tests the hypothesis that two or more population means are equal. ANOVA is useful in experiments with multiple treatments to determine if there are significant differences between them (Wikipedia, 2024a).

5. **Kruskal-Wallis Test**: The Kruskal-Wallis test is a non-parametric method for testing whether samples originate from the same distribution. It extends the Mann-Whitney U test to more than two groups and is equivalent to one-way ANOVA for ranks. It is used to determine if there are statistically significant differences between three or more independent groups (Wikipedia, 2024d).

6. **Post-hoc Analysis (Dunn's Test)**: Post-hoc analysis, such as Dunn's Test, is conducted after an initial statistical test indicates significant differences among group means. Dunn's Test is specifically used following a Kruskal-Wallis test to pinpoint which groups differ. It adjusts for multiple comparisons to control the Type I error rate, ensuring the reliability of the findings by considering the number of comparisons made (Wikipedia, 2024e).

The Python script uses the following libraries to perform the analysis:

1. **pandas v2.0.3**: Provides fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data easy and intuitive (The pandas development team, 2024).

2. **seaborn v0.12.2**: A Python data visualization library based on matplotlib, providing a high-level interface for drawing attractive and informative statistical graphics (Waskom, 2024).

3. **matplotlib v3.7.1**: A state-based interface to matplotlib, providing a MATLAB-like way of plotting (The Matplotlib development team, 2022).

4. **scipy v1.11.1**:

    - **scipy.stats.kruskal**: Computes the Kruskal-Wallis H-test for independent samples (The SciPy community, 2024b).

- **scipy.stats.f_oneway**: Performs one-way ANOVA (The SciPy community, 2024a).
- **scipy.stats.shapiro**: Performs the Shapiro-Wilk test for normality (The SciPy community, 2024c).

5. **scikit-posthocs v0.7.0**: Provides post hoc tests for pairwise multiple comparisons to assess the differences between group levels if a statistically significant result of ANOVA test has been obtained (Maksim Terpilovskii, 2021).

The script performs the following steps:

1. Data Loading: Reads the CSV file into a DataFrame.
2. Output Directory Creation: Ensures an output directory exists.
3. Descriptive Statistics: Computes and saves descriptive statistics.
4. Visualization: Creates and saves boxplots for each metric.
5. Normality Testing: Conducts Shapiro-Wilk tests for normality.
6. ANOVA and Kruskal-Wallis Tests: Performs these tests to check for differences between implementations.
7. Post-hoc Analysis: Conducts post-hoc analysis if the tests show significant differences.
8. Endpoint Analysis: Analyzes data for each endpoint in the dataset.
9. Result Output: Saves the results of all analyses to a text file and images to the output directory.

The script is structured to load the data, generate descriptive statistics, visualize data distributions, test for normality, perform hypothesis testing, and conduct post-hoc analysis where necessary. The results of these analyses are systematically saved into output files and visualizations, providing a thorough examination of the performance metrics.

It begins by importing necessary libraries including `pandas` for data manipulation, `seaborn` and `matplotlib` for visualization, `scipy.stats` for statistical testing, and `scikit_posthocs` for post-hoc analysis. The data is loaded from a CSV file using the `load_data` function illustrated in Listing 28, which utilizes the `pandas.read_csv` method to read the data into a DataFrame.

```python
def load_data(file_path):
    """Load CSV data into a DataFrame."""
    return pd.read_csv(file_path)
```

Listing 28: `load_data` method

To ensure that all output files and visualizations are stored in an organized manner, the script creates an output directory if it does not already exist. This is handled by the `create_output_dir` function using `os.makedirs`:

```python
def create_output_dir(output_dir):
    """Create output directory if it does not exist."""
    os.makedirs(output_dir, exist_ok=True)
```

Listing 29: `create_output_dir` method

The `save_descriptive_statistics` function computes these statistics using the `DataFrame.describe` method and writes them to an output file, as shown in Listing 30.

```python
def save_descriptive_statistics(df, file, level="overall"):
    """Save descriptive statistics to the file."""
    desc_stats = df.describe()
    file.write("\n---\n")
    file.write(f"// descriptive_statistics ({level})\n\n")
    file.write(desc_stats.to_string())
    file.write("\n---\n")
```

Listing 30: `save_descriptive_statistics` method

To visualize the distribution of performance metrics across different implementations, the script generates boxplots using the `create_boxplot` function. These visualizations help in understanding the spread and potential outliers in the data.

```python
def create_boxplot(df, metric, endpoint, output_dir):
    """Create and save a boxplot for the specified metric by implementation."""
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='implementation', y=metric, data=df)
    plt.title(
      f'{metric.replace("_", " ").title()} by Implementation for {endpoint}')
    plot_path = os.path.join(output_dir, f'boxplot_{metric}_{endpoint}.png')
    plt.savefig(plot_path)
    plt.close()
```

Listing 31: `create_boxplot` method

The script performs Shapiro-Wilk tests to determine whether the data follows a normal distribution. These are done for each performance metric using the `perform_normality_tests` function.

```python
def perform_normality_tests(df, metrics, file):
    """Perform Shapiro-Wilk test for normality and save results."""
    file.write("// normality_tests\n\n")
    for metric in metrics:
        file.write(f"Normality tests for {metric}:\n")
        for impl in df['implementation'].unique():
            stat, p = shapiro(df[df['implementation'] == impl][metric])
            file.write(f"  Implementation: {impl}, p-value: {p}\n")
        file.write("\n")
    file.write("\n---\n")
```

Listing 32: `perform_normality_tests` method

To compare the performance metrics across different implementations, the script employs ANOVA (Analysis of Variance) and Kruskal-Wallis tests using the `perform_anova_kruskal_tests` function. These tests assess whether there are statistically significant differences between the means of the groups.

```python
def perform_anova_kruskal_tests(df, metrics, file):
    """Perform ANOVA and Kruskal-Wallis tests and save results."""
    file.write("// anova_kruskal_results\n\n")
    results = {}
    for metric in metrics:
        implementations = df['implementation'].unique()
        groups = [df[df['implementation'] == impl][metric]
                  for impl in implementations]
        anova_result = f_oneway(*groups)
        kruskal_result = kruskal(*groups)
        file.write(f"{metric}:\n")
        file.write(f"  ANOVA result: F-statistic = {anova_result.statistic},
                    p-value = {anova_result.pvalue}\n")
        file.write(f"  Kruskal-Wallis result: H-statistic =
                    {kruskal_result.statistic},
                      p-value = {kruskal_result.pvalue}\n\n")
        results[metric] = (anova_result, kruskal_result)
    file.write("\n---\n")
    return results
```

Listing 33: `perform_anova_kruskal_tests` method

When the ANOVA or Kruskal-Wallis tests indicate significant differences, the script conducts post-hoc analysis using the `perform_posthoc_analysis` function. This analysis, specifically Dunn's test with Bonferroni correction, identifies which specific groups differ from each other.

```python
def perform_posthoc_analysis(df, metric, anova_p, kruskal_p, file):
    """Perform post-hoc analysis if
       ANOVA or Kruskal-Wallis test is significant."""
    if anova_p < 0.05 or kruskal_p < 0.05:
        posthoc = sp.posthoc_dunn(
            df, val_col=metric, group_col='implementation',
            p_adjust='bonferroni'
        )
        file.write(f"// posthoc_analysis ({metric})\n\n")
        file.write(posthoc.to_string())
        file.write("\n---\n")
    else:
        file.write(f"// posthoc_analysis ({metric})\n\n")
        file.write(
            f"{metric}: ANOVA and Kruskal-Wallis tests did not "
            "show significant differences, skipping post-hoc analysis.\n"
        )
        file.write("\n---\n")
```

Listing 34: `perform_posthoc_analysis` method

The `analyze_endpoint` function orchestrates the analysis for each endpoint, invoking the aforementioned functions to perform descriptive statistics, visualizations, normality tests, hypothesis testing, and post-hoc analysis. It is shown in Listing 35:

```python
def analyze_endpoint(df, endpoint, output_file, output_dir):
    """Perform analysis for a specific endpoint."""
    df_endpoint = df[df['endpoint_name'] == endpoint]

    with open(output_file, "a") as file:
        file.write(f"\n---\nEndpoint: {endpoint}\n---\n")

        # Overall Descriptive Statistics
        save_descriptive_statistics(df_endpoint, file, level="overall")

        # Descriptive Statistics by Implementation
        for impl in df_endpoint['implementation'].unique():
            df_impl = df_endpoint[df_endpoint['implementation'] == impl]
            save_descriptive_statistics(df_impl, file, level=impl)

        # Visualization
        metrics = ['req_dur_avg_ms', 'req_dur_p_90_ms', 'req_dur_p_95_ms']
        for metric in metrics:
            create_boxplot(df_endpoint, metric, endpoint, output_dir)

        # Normality Tests
        perform_normality_tests(df_endpoint, metrics, file)

        # ANOVA and Kruskal-Wallis Tests
        results = perform_anova_kruskal_tests(df_endpoint, metrics, file)

        # Post-hoc Analysis
        for metric, (anova_result, kruskal_result) in results.items():
            perform_posthoc_analysis(
                df_endpoint, metric,
                anova_result.pvalue, kruskal_result.pvalue,
                file)
```

Listing 35: `analyze_endpoint` method

The script's `main` function main loads the data, creates the output directory, and iterates over each endpoint to perform the analyses. The results are saved to an output directory, ensuring all outputs are systematically organized.

```python
def main():
    # Load data
    print("Step 1: Loading data...")
    df = load_data('k6_performance_data.csv')

    # Create output directory
    output_dir = "analysis_results"
    create_output_dir(output_dir)

    # Open the consolidated output file
    output_file = os.path.join(output_dir, "analysis_results.txt")

    # Analyze each endpoint
    for endpoint in df['endpoint_name'].unique():
        analyze_endpoint(df, endpoint, output_file, output_dir)

    print("Analysis complete. \
          Results have been saved to the 'analysis_results' directory.")
```

Listing 36: `main` method

By following this structured approach, the script ensures that each step of the data analysis process is methodically executed and documented, providing insights into the performance data across different implementations.

6 Results

This chapter presents the findings from the load testing of the Traditional API and the CQRS-implemented APIs. In Appendix A, tables for the descriptive statistics of each endpoint and implementation are provided, totaling 27 tables. These tables describe the dataset analyzed. The descriptive statistics table summarizes key metrics of the data, providing a comprehensive overview of the dataset's distribution and central tendencies. Below is an example table for the `GetCategoryMapping` endpoint using the `TraditionalApi` implementation. Each row and column in the table is defined as follows:

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1361.23	2.55	15.21	6.76	9.58	0.0
std	0.43	0.11	2.06	0.23	0.31	0.0
min	1361.00	2.28	10.97	6.36	9.04	0.0
max	1362.00	2.72	19.61	7.12	10.00	0.0

Table 2: GetCategoryMapping: Descriptive statistics (TraditionalApi)

Table Columns:

- **reqs_count:** The number of requests processed.

- **req_dur_min_ms:** The minimum request duration in milliseconds.
- **req_dur_max_ms:** The maximum request duration in milliseconds.
- **req_dur_avg_ms:** The average request duration in milliseconds.
- **req_dur_p_95_ms:** The 95th percentile request duration in milliseconds.
- **req_failed:** The number of failed requests.

Table Rows:

- **count:** The number of data points (requests) used for each statistic.
- **mean:** The average value for each metric.
- **std:** The standard deviation, showing the variation or dispersion of the data from the mean.
- **min:** The minimum value observed for each metric.
- **max:** The maximum value observed for each metric.

Interpreting the Table:

- The **count** row shows that there are 30 data points for each metric, indicating that 30 requests were analyzed.
- The **mean** row provides the average values: for example, the average request count is 1361.23, and the average request duration is 6.76 milliseconds.
- The **std** (standard deviation) row indicates the variability in the data: for example, the request duration (average) has a standard deviation of 0.23 milliseconds, suggesting low variability around the mean.
- The **min** and **max** rows show the range of values observed: for example, request durations range from 2.28 to 19.61 milliseconds.

By examining these statistics, one can understand the performance characteristics and stability of the `TraditionalApi` implementation for the `GetCategoryMapping` endpoint. Similar tables for other endpoints and implementations can be interpreted in the same manner.

In Appendix B, there are 18 box plots, each representing an endpoint and all three implementations for one of the metrics `req_dur_avg_ms` and `req_dur_p_95_ms`. The results of the `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` endpoints will be explained in detail in the following subsections.

In performance analysis, using the 95th percentile instead of the average is preferable because the average can be significantly affected by outliers, which can distort the true performance picture. Percentile

ranks, such as the 95th percentile, provide a more robust and accurate measure by indicating the performance level below which 95% of the observations fall. This approach is less influenced by extreme values and better reflects the typical experience or performance within a distribution, as highlighted by Bornmann and Williams in their evaluation of citation impact measures (Bornmann & Williams, 2020). Therefore, the average is displayed in the tables but will not be discussed; rather, the analysis focuses on the 95th percentile.

In the next paragraphs, the discussion of the Shapiro-Wilk (Normality), ANOVA, Kruskal-Wallis, and Posthoc test results for all endpoints and implementations will be presented.

6.1 Shapiro-Wilk (Normality)

Based on the Shapiro-Wilk test results in Table 3, the normality of the data for different endpoints and implementations can be interpreted. The Shapiro-Wilk test checks the null hypothesis that the data is normally distributed. A p-value less than the chosen alpha level of 0.05 indicates that the null hypothesis can be rejected, suggesting that the data is not normally distributed (The SciPy community, 2024c; Wikipedia, 2024f).

The data for most endpoints and implementations do not follow a normal distribution, as indicated by the p-values being less than 0.05. Only the data for the following endpoints are normally distributed:

- GetLeafAttributes (all implementations)
- GetSubAttributes (TraditionalApi and CqrsApiMediator)
- GetChildrenOrTopLevel (CqrsApi and CqrsApiMediator)
- SearchCategories (TraditionalApi)
- UpdateCategoryMapping (TraditionalApi)
- GetRootCategories (TraditionalApi and CqrsApi)

Endpoint	Implementation	Shapiro-Wilk req_dur_avg_ms (p-value)	Shapiro-Wilk req_dur_p_95_ms (p-value)
GetAttributes	TraditionalApi	2.0708994270535186e-06	9.906215154842357e-07
GetAttributes	CqrsApi	0.020385434851050377	0.004787254147231579
GetAttributes	CqrsApiMediator	0.05882410332560539	0.00012437465193215758
GetLeafAttributes	TraditionalApi	0.08619271218776703	0.7344048023223877
GetLeafAttributes	CqrsApi	0.707417905330658	0.30365607142448425
GetLeafAttributes	CqrsApiMediator	0.2119186967611313	0.42768946290016174
GetSubAttributes	TraditionalApi	0.03823234885931015	0.2277591973543167
GetSubAttributes	CqrsApi	0.08713330328464508	0.03528261557221413
GetSubAttributes	CqrsApiMediator	0.8599060773849487	0.09247878193855286
UpdateAttributeValues	TraditionalApi	0.00018826367158908397	0.0003938550071325153
UpdateAttributeValues	CqrsApi	6.5217218434554525e-06	0.0002098339464282617
UpdateAttributeValues	CqrsApiMediator	3.7325315815905924e-07	1.6862346456036903e-05
GetCategoryMapping	TraditionalApi	0.016934312880039215	0.0050710225477814674
GetCategoryMapping	CqrsApi	0.022493524476885796	0.03988869860768318
GetCategoryMapping	CqrsApiMediator	0.009106666781008244	0.001869262894615531
GetChildrenOrTopLevel	TraditionalApi	1.0662335370170695e-07	1.4464320408080766e-08
GetChildrenOrTopLevel	CqrsApi	0.4232797920703888	0.3918805420398712
GetChildrenOrTopLevel	CqrsApiMediator	0.0856046974658966	0.6819615960121155
SearchCategories	TraditionalApi	0.007911057211458683	0.05256574973464012
SearchCategories	CqrsApi	0.047641362994909286	8.818412607070059e-05
SearchCategories	CqrsApiMediator	0.00034235589555464685	1.948035333043663e-06
UpdateCategoryMapping	TraditionalApi	0.6703578233718872	0.2554927468299866
UpdateCategoryMapping	CqrsApi	0.010161234997212887	0.01644904352724552
UpdateCategoryMapping	CqrsApiMediator	0.12713773548603058	0.044561926275491714
GetRootCategories	TraditionalApi	0.3039284348487854	0.971557080745697
GetRootCategories	CqrsApi	0.03522524982690811	0.15877313911914825
GetRootCategories	CqrsApiMediator	4.3258332027562574e-08	1.2915626901310873e-10

Table 3: Shapiro-Wilk Test Results for req_dur_avg_ms and req_dur_p_95_ms across Endpoints and Implementations

6.2 ANOVA

The ANOVA test results in Table 4 provide insights into whether there are statistically significant differences in the request duration (95th percentile in milliseconds) across different implementations for each endpoint. The ANOVA test checks the null hypothesis that there are no differences among the group means. A p-value less than the chosen alpha level of 0.05 indicates that the null hypothesis can be rejected, suggesting that there are significant differences among the group means (Wikipedia, 2024a). The ANOVA test indicates that there are significant differences in request durations for the `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` endpoints across different implementations, as evidenced by their p-values being less than 0.05. For the other endpoints, the p-values are greater than 0.05, indicating no significant differences in request durations among the different implementations.

Endpoint	ANOVA req_dur_p_95_ms (F-statistic)	ANOVA req_dur_p_95_ms (p-value)
GetAttributes	0.41780424583110515	0.6598052519849145
GetLeafAttributes	1.387812023535741	0.25509131215198094
GetSubAttributes	1.4796248769558826	0.23339637622043155
UpdateAttributeValues	0.6309852490892742	0.5344844302744204
GetCategoryMapping	69.01034178495553	1.1153504938177082e-18
GetChildrenOrTopLevel	9.52417292876415	0.00018183098528554858
SearchCategories	17.467354043173906	4.1932240950866433e-07
UpdateCategoryMapping	2.2678069403229912	0.10962717065213491
GetRootCategories	0.9922657829860185	0.3748917915932988

Table 4: ANOVA Test Results for req_dur_p_95_ms across Endpoints

Endpoint	ANOVA req_dur_avg_ms (F-statistic)	ANOVA req_dur_avg_ms (p-value)
GetAttributes	0.7675235249386696	0.46727797767785706
GetLeafAttributes	0.5529003248390815	0.5772868248723405
GetSubAttributes	1.1290797456058475	0.3280217537717261
UpdateAttributeValues	2.493227715097005	0.08853164300934498
GetCategoryMapping	42.79582892678614	1.1447909029922702e-13
GetChildrenOrTopLevel	17.508816490146316	4.0710324953372236e-07
SearchCategories	43.3941249628587	8.476170057007661e-14
UpdateCategoryMapping	1.0588444661073808	0.35128302485783225
GetRootCategories	1.416469245393784	0.24810677860286845

Table 5: ANOVA Test Results for req_dur_avg_ms across Endpoints

6.3 Kruskal-Wallis

The Kruskal-Wallis test results in Table 6, show whether there are statistically significant differences in the request duration (95th percentile in milliseconds) across different implementations for each endpoint. The Kruskal-Wallis test is a non-parametric method used to test whether there are statistically significant differences between the distributions of three or more independent groups. A p-value less than the chosen alpha level of 0.05 indicates that the null hypothesis (that the distributions are the same) can be rejected, suggesting significant differences among the group distributions (Wikipedia, 2024d). The Kruskal-Wallis test indicates that there are significant differences in request durations for the `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` endpoints across different implementations, as evidenced by their p-values being less than 0.05. For the other endpoints, the p-values are greater than 0.05, indicating no significant differences in request durations among the different implementations.

Endpoint	Kruskal-Wallis req_dur_p_95_ms (H-statistic)	Kruskal-Wallis req_dur_p_95_ms (p-value)
GetAttributes	0.9107692307691764	0.6342039964196438
GetLeafAttributes	1.7031501831502283	0.4267422442018113
GetSubAttributes	3.308522588522578	0.19123326865462276
UpdateAttributeValues	5.23555555555555	0.07296482686043715
GetCategoryMapping	54.03194139194136	1.849749861517264e-12
GetChildrenOrTopLevel	59.875555555555536	9.958371200518601e-14
SearchCategories	59.341050061050055	1.300932322266575e-13
UpdateCategoryMapping	3.097142857142842	0.21255140190066082
GetRootCategories	1.1828083028082688	0.5535494715108595

Table 6: Kruskal-Wallis Test Results for req_dur_p_95_ms across Endpoints

Endpoint	Kruskal-Wallis req_dur_avg_ms (H-statistic)	Kruskal-Wallis req_dur_avg_ms (p-value)
GetAttributes	0.07238095238096776	0.9644565699728658
GetLeafAttributes	0.5809035409035346	0.7479256005561047
GetSubAttributes	3.1393406593406894	0.2081137799875518
UpdateAttributeValues	2.69216117216115	0.2602583243351654
GetCategoryMapping	44.580903540903535	2.0863145985910806e-10
GetChildrenOrTopLevel	59.21084249084248	1.3884457603297446e-13
SearchCategories	59.34222222222223	1.3001700944739522e-13
UpdateCategoryMapping	2.0044932844933214	0.3670538753999452
GetRootCategories	0.7485225885225759	0.6877971708936801

Table 7: Kruskal-Wallis Test Results for req_dur_avg_ms across Endpoints

6.4 Posthoc

The Posthoc analysis is only performed when the ANOVA or Kruskal-Wallis tests indicate significant differences. Therefore, only the endpoints `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` are tested. The results are shown in Table 8 for the 95th percentile metric.

The Posthoc test results in Table 8 present the significant pairwise differences in request durations (95th percentile in milliseconds) for the three endpoints `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` across different implementations (Maksim Terpilovskii, 2021). The Posthoc analysis is conducted following significant results from ANOVA or Kruskal-Wallis tests to determine which specific groups are significantly different from each other.

For the endpoints `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories`, the Posthoc test results indicate that:

- There are significant differences in request durations between TraditionalApi and both CqrsApi and CqrsApiMediator implementations.
- There are no significant differences in request durations between CqrsApi and CqrsApiMediator implementations.

These results suggest that the TraditionalApi implementation significantly differs in performance compared to the CqrsApi and CqrsApiMediator implementations for the specified endpoints, while CqrsApi and CqrsApiMediator are similar in performance.

Endpoint	Implementation	TraditionalApi	CqrsApi	CqrsApiMediatr
GetCategoryMapping	TraditionalApi	1.000000e+00	9.309270e-11	4.383449e-09
GetCategoryMapping	CqrsApi	9.309270e-11	1.000000e+00	1.000000e+00
GetCategoryMapping	CqrsApiMediator	4.383449e-09	1.000000e+00	1.000000e+00
GetChildrenOrTopLevel	TraditionalApi	1.000000e+00	8.614565e-10	5.895779e-12
GetChildrenOrTopLevel	CqrsApi	8.614565e-10	1.000000e+00	1.000000e+00
GetChildrenOrTopLevel	CqrsApiMediator	5.895779e-12	1.000000e+00	1.000000e+00
SearchCategories	TraditionalApi	1.000000e+00	7.113340e-11	8.139126e-11
SearchCategories	CqrsApi	7.113340e-11	1.000000e+00	1.000000e+00
SearchCategories	CqrsApiMediator	8.139126e-11	1.000000e+00	1.000000e+00

Table 8: Posthoc Analysis Results for req_dur_p_95_ms across Endpoints and Implementations

Endpoint	Implementation	TraditionalApi	CqrsApi	CqrsApiMediatr
GetCategoryMapping	TraditionalApi	1.000000e+00	1.674779e-09	4.736937e-07
GetCategoryMapping	CqrsApi	1.674779e-09	1.000000e+00	1.000000e+00
GetCategoryMapping	CqrsApiMediator	4.736937e-07	1.000000e+00	1.000000e+00
GetChildrenOrTopLevel	TraditionalApi	1.000000e+00	9.309270e-11	6.877357e-11
GetChildrenOrTopLevel	CqrsApi	9.309270e-11	1.000000e+00	1.000000e+00
GetChildrenOrTopLevel	CqrsApiMediator	6.877357e-11	1.000000e+00	1.000000e+00
SearchCategories	TraditionalApi	1.000000e+00	8.704974e-11	6.649043e-11
SearchCategories	CqrsApi	8.704974e-11	1.000000e+00	1.000000e+00
SearchCategories	CqrsApiMediator	6.649043e-11	1.000000e+00	1.000000e+00

Table 9: Posthoc Analysis Results for req_dur_avg_ms across Endpoints and Implementations

6.5 Box Plots

The next three subsections present box plots comparing the request duration at the 95th percentile (`req_dur_p_95_ms`) for the `GetCategoryMapping`, `GetChildrenOrTopLevel`, and `SearchCategories` endpoints across three different implementations: `TraditionalApi`, `CqrsApi`, and `CqrsApiMediatr`. Here's a detailed explanation of the box plot:

- **Box**: Represents the interquartile range (IQR), which is the range between the first quartile (Q1, 25th percentile) and the third quartile (Q3, 75th percentile).
- **Median Line**: The line inside the box indicates the median (50th percentile).
- **Whiskers**: The lines extending from the box represent the range of the data, excluding outliers. They typically extend to 1.5 times the IQR from the quartiles.
- **Outliers**: Data points outside the whiskers, not present in this plot.

6.5.1 GetCategoryMapping

- **TraditionalApi**:
  - The median request duration at the 95th percentile is just below 9.75 ms.
  - The IQR ranges approximately from 9.5 ms to 10 ms.
  - The whiskers extend from around 9 ms to slightly above 10 ms, indicating the spread of the majority of the data points.
- **CqrsApi**:
  - The median request duration is about 10.25 ms.
  - The IQR ranges approximately from 10 ms to 10.75 ms.
  - The whiskers extend from slightly above 9.75 ms to around 11 ms, showing a wider spread compared to TraditionalApi.
- **CqrsApiMediatr**:

- The median request duration is slightly below 10.5 ms.
- The IQR ranges approximately from 10 ms to 10.75 ms.
- The whiskers extend from around 9.75 ms to just below 11 ms, similar in spread to CqrsApi.

In summary, the `TraditionalApi` has the lowest median request duration, indicating better performance at the 95th percentile. The `CqrsApi` and `CqrsApiMediatr` have similar median values and IQRs, suggesting they have comparable performance, though their medians are higher than the TraditionalApi. The spread (represented by the whiskers) is slightly wider for `CqrsApi` and `CqrsApiMediatr`, indicating more variability in their request durations compared to `TraditionalApi`.

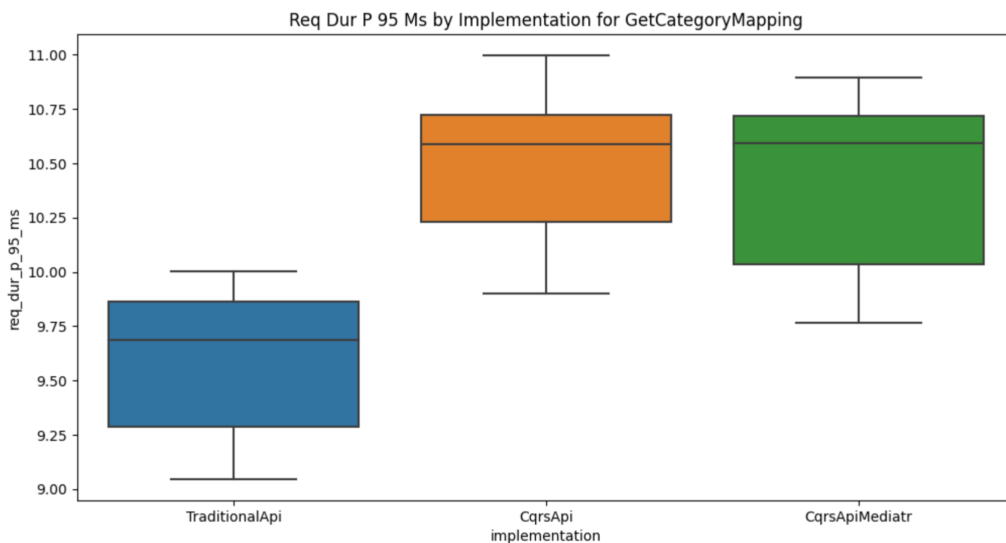

Figure 1: GetCategoryMapping 95th Percentile in milliseconds

6.5.2 GetChildrenOrTopLevel

- **TraditionalApi**:
  - The median request duration at the 95th percentile is slightly above 150 ms.
  - The IQR ranges approximately from 100 ms to 200 ms.
  - The whiskers extend from around 50 ms to 225 ms.
  - Several outliers are present, with request durations significantly higher, some close to 1000 ms.
- **CqrsApi**:
  - The median request duration is slightly below 20 ms.
  - The IQR ranges approximately from 15 ms to 25 ms.
  - The whiskers extend from around 10 ms to 30 ms.
  - A few outliers are present, but they are relatively close to the upper whisker.
- **CqrsApiMediatr**:

- The median request duration is also slightly below 20 ms.
- The IQR ranges approximately from 15 ms to 25 ms.
- The whiskers extend from around 10 ms to 30 ms.
- Similar to CqrsApi, a few outliers are present but relatively close to the upper whisker.

In summary, the `TraditionalApi` shows significantly higher median and variability in request duration, with multiple extreme outliers. Both `CqrsApi` and `CqrsApiMediatr` have much lower and more consistent request durations, with similar median values and IQRs. The spread (represented by the whiskers) and the presence of outliers in `TraditionalApi` indicate less reliable performance compared to `CqrsApi` and `CqrsApiMediatr`.

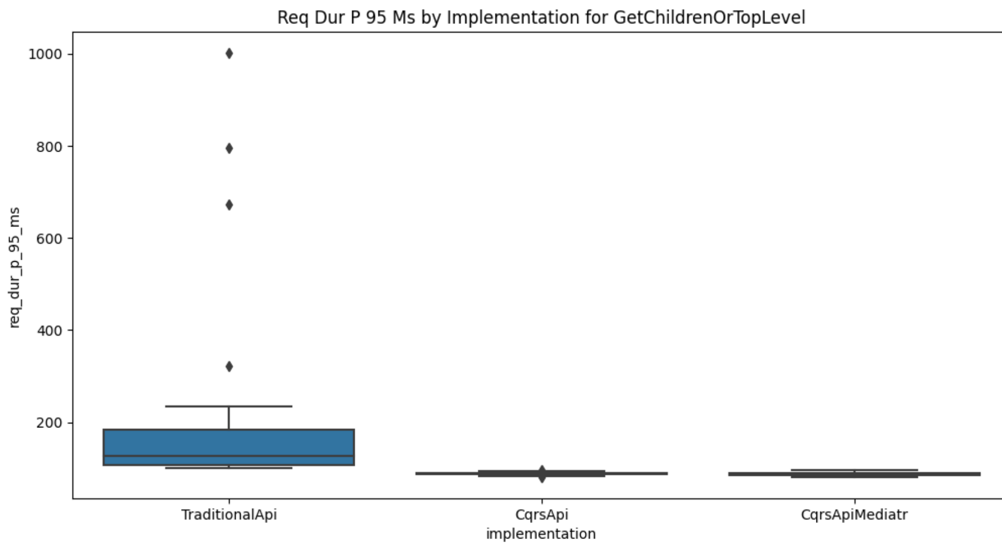

Figure 2: GetChildrenOrTopLevel 95th Percentile in milliseconds

6.5.3 SearchCategories

- **TraditionalApi**:
  - The median request duration at the 95th percentile is around 50 ms.
  - The IQR is very narrow, indicating low variability.
  - The whiskers extend slightly beyond the box, indicating very consistent performance.
  - There is one minor outlier.
- **CqrsApi**:
  - The median request duration is around 75 ms.
  - The IQR ranges approximately from 50 ms to 100 ms.
  - The whiskers extend from around 25 ms to 150 ms, showing more variability.
  - Several outliers are present, with request durations up to around 250 ms.
- **CqrsApiMediatr**:

– The median request duration is around 70 ms.
– The IQR ranges approximately from 50 ms to 100 ms.
– The whiskers extend from around 25 ms to 150 ms.
– Similar to CqrsApi, several outliers are present, though they are relatively fewer and lower than those of CqrsApi.

In summary, `TraditionalApi` shows the lowest median and variability in request duration, indicating very consistent and fast performance. Both `CqrsApi` and `CqrsApiMediatr` have higher and more variable request durations compared to `TraditionalApi`. The presence of outliers in `CqrsApi` and `CqrsApiMediatr` indicates occasional high request durations, affecting overall performance consistency.

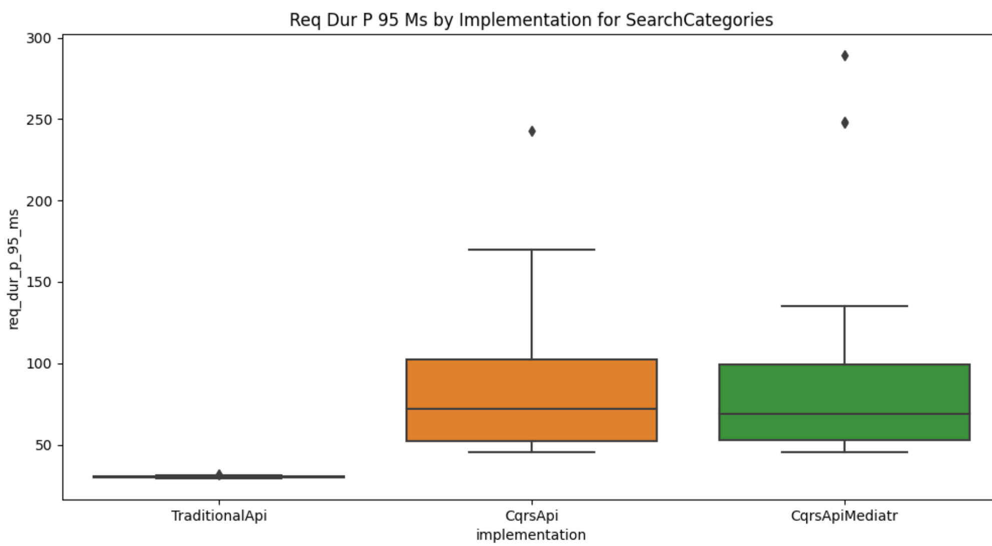

Figure 3: SearchCategories 95th Percentile in milliseconds

7 Conclusion

This thesis aimed to evaluate whether different implementations of the Command and Query Responsibility Segregation (CQRS) pattern could significantly improve the response times of a Traditional API. The study compared the performance of a Traditional API against two CQRS-based implementations: one manually implemented and the other using the MediatR library (Bogard, n.d.). Performance was assessed using load testing, statistical analysis, and various endpoint metrics to determine the impact of CQRS on response times. The study provides the following key findings:

Performance Analysis

- Significant differences in request durations were observed for three endpoints:
  - **GetCategoryMapping**: The Traditional API performed slightly but significantly better than both CQRS implementations.
  - **GetChildrenOrTopLevel**: Both CQRS implementations outperformed the Traditional API significantly.
  - **SearchCategories**: The Traditional API performed significantly better than both CQRS implementations.
- For other endpoints, there were no significant differences in performance between the Traditional API and the CQRS implementations. This suggests that the CQRS implementations used in the study do not overall provide a significant performance improvement over the Traditional API.
- The MediatR-based CQRS implementation does not show significant differences from the manually implemented CQRS implementation in terms of performance, according to the Posthoc analysis.

Potential Reasons for Performance Results

- The lack of significant performance improvement could be attributed to several factors:
  - The CQRS implementations might not be fully optimized.
  - The Traditional API could already be highly efficient, leaving little room for noticeable improvement with CQRS.
- Further investigation is needed to determine the cause of these performance differences. Implementing CQRS in combination with Domain-Driven Design (DDD) and Event Sourcing (ES), and using separate databases for read and write operations, could potentially offer better performance improvements (Young, 2010, p. 50).

The findings suggest that while CQRS can offer structural benefits in terms of clear separation of read and write operations, the overall performance gains in this study were not significant. This study could not evaluate whether or not the CQRS enhances maintainability. Greg Young mentions that: "The Thin Read Layer is not a complex piece of code although it can be tedious to maintain" (Young, 2010, p. 21).

However, he also promotes the benefits of the Thin Read Layer: "It is connected directly to the data model, this can make queries much easier to optimize. Developers working on the Query side of the system also do not need to understand the domain model nor whatever ORM tool is being used. At the simplest level they would need to understand only the data model. The separation of the Thin Read Layer and the bypassing of the domain for reads allows also for the specialization of the domain" (Young, 2010, pp. 21–22).

When considering the adoption of CQRS, organizations should weigh the potential benefits against the actual performance observed for the specific use case and the requirements of their system. When looking to optimize performance with CQRS, it may be beneficial to first ensure that the current system is fairly optimized and, when implementing CQRS, to consider the integration of Domain-Driven Design and Event Sourcing for a more comprehensive approach (Young, 2010, p. 50). Additionally, performance optimizations applied to the original Traditional API might yield similar improvements without the added complexity or cost of restructuring and refactoring the project to a CQRS architecture. For established systems, the cost of migrating to Domain-Driven Design (DDD) and Event Sourcing (ES) only to get the full benefits of CQRS might not be worth the effort if the current system is already performing well enough (Fowler et al., 1999, p. 54).

Future research should explore the combined impact of CQRS with DDD and ES to determine if these patterns together can provide more significant performance benefits. Further studies should also investigate other performance metrics, such as throughput and resource utilization, to provide a more comprehensive understanding of CQRS benefits. Additionally, extending the K6 duration to more than 2 minutes and using a different virtual user count could provide more data points and help verify the consistency of the results. Using metric data directly for statistical analysis instead of relying on the summary export calculated by K6 could offer deeper insights into outliers. Moreover, the CQRS implementation on the read side could be optimized by using a read replica database.

In conclusion, the implementation of CQRS showed slight performance improvements in one case but did not provide overall significant gains compared to the Traditional API. Further optimization and research are required to investigate the potential performance benefits of CQRS in real-world applications.

Eidesstattliche Erklärung

Ich, Jens Richter, erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen, als die von mir angegebenen Quellen und Hilfsmittel (inklusive elektronischer Medien und Online-Ressourcen) benutzt. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass ein Verstoß gegen diese Versicherung nicht nur prüfungsrechtliche Folgen haben wird, sondern auch zu weitergehenden rechtlichen Konsequenzen führen kann.

Leipzig, 25. July 2024

____

Jens Richter

Listings

1 Hello World as a minimal API 4

2 Hello World using a Controller 5

3 The `HelloWorldController` 5

4 Example JSON Response from `/categories` 6

5 Example Command `DeactivateInventoryItemCommand` 12

6 Original Customer Service (Young, 2010, p. 18) 13

7 Customer Service after CQRS (Young, 2010, p. 19) 13

8 `CqrsReadDbContext` protection 23

9 Traditional API structure 24

10 CQRS API structure 24

11 `SearchCategoriesEndpoint` with MediatR 26

12 `SearchCategoriesQuery` with MediatR 26

13 `SearchCategoriesQueryHandler` with MediatR 27

14 Dependency Injection: MediatR handlers 27

15 `K6-GetAttributes` script 28

16 `StartAllFinalK6TestAsync` 31

17 `GetAllTestInfosPerEndpointForApi` method 32

18 `TestInformation` 33

19 `TestInformation` constructor 34

20 `CreateInfoForGetAttributes` method in `TestInformation` 34

21 `K6TestHandler` constructor 35

22 `StartK6TestAndProcessResultsAsync` method - Part 1 36

23 `StartK6TestAndProcessResultsAsync` method - Part 2 36

24 `StartK6TestAndProcessResultsAsync` method - Part 3 37

25 `StartK6TestAndProcessResultsAsync` method - Part 4 37

26 StartK6TestAndProcessResultsAsync method - Part 5 38

27 ThrowIfRequiredContainersAreNotRunningAsync method 39

28 load_data method 49

29 create_output_dir method 50

30 save_descriptive_statistics method 50

31 create_boxplot method 50

32 perform_normality_tests method 51

33 perform_anova_kruskal_tests method 51

34 perform_posthoc_analysis method 52

35 analyze_endpoint method 53

36 main method 54

References

Bogard, J. (n.d.). *Mediatr*. Retrieved June 10, 2024, from https://github.com/jbogard/MediatR

Bogard, J. (2024). *Mediatr*. Retrieved July 14, 2024, from https://www.nuget.org/packages/MediatR/

Bornmann, L., & Williams, R. (2020). An evaluation of percentile measures of citation impact, and a proposal for making them better. *Scientometrics*, *125*(2), 1033–1052.

Docker. (2024). *Docker overview*. Retrieved June 10, 2024, from https://docs.docker.com/guides/docker-overview/

Ecma International. (2017). *Ecma-404: The json data interchange syntax*. Retrieved June 17, 2024, from https://ecma-international.org/publications-and-standards/standards/ecma-404/

Elasticsearch. (n.d.). *Meet the search platform that helps you search, solve, and succeed*. Retrieved June 10, 2024, from https://www.elastic.co/elastic-stack/

Evans, E. (2014). *Domain-driven design reference: Definitions and pattern summaries*. Dog Ear Publishing.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* [Doctoral dissertation, University of California, Irvine]. Retrieved July 14, 2024, from https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Fowler, M. (2005). *Domain event*. Retrieved July 14, 2024, from https://martinfowler.com/eaaDev/DomainEvent.html

Fowler, M. (2010). *Richardson maturity model*. Retrieved July 14, 2024, from https://martinfowler.com/articles/richardsonMaturityModel.html

Fowler, M. (2011). *Cqrs*. Retrieved June 10, 2024, from https://martinfowler.com/bliki/CQRS.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code* (J. C. Shanklin, Ed.) [With contributions by Kent Beck, John Brant, et al. Foreword by Erich Gamma]. Addison-Wesley.

Grafana Labs. (n.d.-a). *Grafana k6*. Retrieved July 14, 2024, from https://grafana.com/docs/k6/latest/

Grafana Labs. (n.d.-b). *Results output*. Retrieved July 14, 2024, from https://grafana.com/docs/k6/latest/get-started/results-output/

Grafana Labs. (n.d.-c). *Running k6*. Retrieved July 14, 2024, from https://grafana.com/docs/k6/latest/get-started/running-k6/

Hibernate. (n.d.). *What is object/relational mapping?* Retrieved June 17, 2024, from https://hibernate.org/orm/what-is-an-orm/

Hofmeister, A., et al. (n.d.). *Welcome to testcontainers for .net!* Retrieved July 14, 2024, from https://dotnet.testcontainers.org/

Maksim Terpilovskii. (2021). *Scikit-posthocs*. Retrieved July 15, 2024, from https://scikit-posthocs.readthedocs.io/en/stable/

Meyer, B. (1997). *Object-oriented software construction* (Vol. 2). Prentice hall Englewood Cliffs.

Microsoft. (2021a). *Entity framework core*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/ef/core/

Microsoft. (2021b). *Lazy loading of related data*. Retrieved July 14, 2024, from https://learn.microsoft.com/en-us/ef/core/querying/related-data/lazy

Microsoft. (2022a). *An introduction to nuget*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/nuget/what-is-nuget

Microsoft. (2022b). *Minimal apis overview*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/overview?view=aspnetcore-8.0

Microsoft. (2023a). *Overview of asp.net core*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0

Microsoft. (2023b). *Records (C# reference)*. Retrieved July 14, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record

Microsoft. (2023c). *Tracking vs. no-tracking queries*. Retrieved July 14, 2024, from https://learn.microsoft.com/en-us/ef/core/querying/tracking

Microsoft. (2024a). *Create web apis with asp.net core*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-8.0

Microsoft. (2024b). *Description of the database normalization basics*. Retrieved July 14, 2024, from https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description

Microsoft. (2024c). *Introduction to .net*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/dotnet/core/introduction

Microsoft. (2024d). *Top-level statements - programs without main methods*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements

Microsoft. (2024e). *A tour of the C# language*. Retrieved June 10, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview

Microsoft. (2024f). *What's new in .net 8*. Retrieved June 16, 2024, from https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview

Microsoft. (2024g). *What's new in C# 12*. Retrieved June 16, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-12

Miller, D., Whitlock, J., Gardiner, M., Ralphson, M., Ratovsky, R., Sarid, U., Harmon, J., & Tam, T. (2021). *Openapi specification v3.1.0*. Retrieved June 17, 2024, from https://spec.openapis.org/oas/latest.html

Npgsql. (n.d.). *Npgsql - .net access to postgresql*. Retrieved June 10, 2024, from https://www.npgsql.org/

PostgreSQL. (2024). *What is postgresql?* Retrieved June 10, 2024, from https://www.postgresql.org/docs/16/intro-whatis.html#INTRO-WHATIS

Relaxdays. (2024). *Develop future commerce*. Retrieved July 14, 2024, from https://relaxdays-unternehmen.de/

Serilog. (n.d.). *Flexible, structured events — log file convenience.* Retrieved June 10, 2024, from https://serilog.net/

The Matplotlib development team. (2022). *Matplotlib.pyplot*. Retrieved July 15, 2024, from https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

The pandas development team. (2024). *Pandas: Powerful python data analysis toolkit*. Retrieved July 15, 2024, from https://github.com/pandas-dev/pandas/tree/v2.2.2

The SciPy community. (2024a). *Scipy.stats.f_oneway*. Retrieved July 15, 2024, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

The SciPy community. (2024b). *Scipy.stats.kruskal*. Retrieved July 15, 2024, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kruskal.html

The SciPy community. (2024c). *Scipy.stats.shapiro*. Retrieved July 15, 2024, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html

van Oudheusden, M. (n.d.). *Serilog.sinks.elasticsearch*. Retrieved July 14, 2024, from https://github.com/serilog-contrib/serilog-sinks-elasticsearch

Vogels, W. (2008). Eventually consistent - revisited. *All Things Distributed*. Retrieved July 14, 2024, from https://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Waskom, M. (2024). *Seaborn: Statistical data visualization*. Retrieved July 15, 2024, from https://seaborn.pydata.org/

Wikipedia. (2023). *You aren't gonna need it*. Retrieved July 14, 2024, from https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

Wikipedia. (2024a). *Analysis of variance*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Analysis_of_variance

Wikipedia. (2024b). *Box plot*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Box_plot

Wikipedia. (2024c). *Descriptive statistics*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Descriptive_statistics

Wikipedia. (2024d). *Kruskal–wallis test*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_test

Wikipedia. (2024e). *Post hoc analysis*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Post_hoc_analysis

Wikipedia. (2024f). *Shapiro–wilk test*. Retrieved July 15, 2024, from https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test

Wikipedia. (2024g). *Vertical slice*. Retrieved July 14, 2024, from https://en.wikipedia.org/wiki/Vertical_slice#:~:text=The%20term%20vertical%20slice%20refers,of%20features%20(or%20stories).

Young, G. (2010). Cqrs documents by greg young. Retrieved June 10, 2024, from https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf

A Tables

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1345.83	4.72	97.35	18.86	27.70	0.0
std	1.11	0.28	128.95	0.97	2.92	0.0
min	1340.00	4.22	33.76	17.69	25.22	0.0
max	1347.00	5.15	653.24	22.90	40.42	0.0

Table 10: GetAttributes: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1346.16	4.76	64.55	18.65	27.76	0.0
std	0.59	0.27	23.75	0.46	2.40	0.0
min	1345.00	4.31	35.81	17.94	24.91	0.0
max	1348.00	5.46	121.61	20.18	34.71	0.0

Table 11: GetAttributes: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1346.10	4.73	60.41	18.69	27.23	0.0
std	0.60	0.24	30.83	0.48	2.09	0.0
min	1345.00	4.21	35.64	17.88	24.96	0.0
max	1348.00	5.24	184.35	19.65	32.48	0.0

Table 12: GetAttributes: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1157.86	5.46	1929.29	186.03	808.41	0.0
std	42.67	0.30	416.77	45.17	198.70	0.0
min	1037.00	4.91	1284.28	112.88	401.90	0.0
max	1233.00	6.07	2788.74	322.62	1201.13	0.0

Table 13: GetLeafAttributes: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1166.53	5.43	1837.04	177.59	755.16	0.0
std	48.96	0.32	417.50	49.79	250.75	0.0
min	1063.00	4.92	1196.85	99.16	282.66	0.0
max	1247.00	6.23	2805.59	291.07	1203.40	0.0

Table 14: GetLeafAttributes: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1153.66	5.41	1999.97	190.50	851.92	0.0
std	47.03	0.28	396.75	49.79	223.45	0.0
min	1042.00	4.84	1349.88	97.64	289.39	0.0
max	1250.00	6.05	2763.76	315.85	1370.27	0.0

Table 15: GetLeafAttributes: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1124.50	6.25	1902.67	220.80	939.06	0.0
std	41.22	0.37	408.32	46.56	167.56	0.0
min	1008.00	5.66	1292.74	144.24	580.85	0.0
max	1198.00	7.01	2976.93	361.27	1292.65	0.0

Table 16: GetSubAttributes: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1126.80	6.30	1834.69	219.78	910.10	0.0
std	51.26	0.36	334.67	55.90	177.11	0.0
min	1023.00	5.62	1297.13	138.66	502.16	0.0
max	1203.00	7.13	2890.17	339.38	1174.30	0.0

Table 17: GetSubAttributes: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1110.53	6.26	1970.81	236.78	983.81	0.0
std	40.30	0.29	357.11	44.24	156.39	0.0
min	1034.00	5.74	1399.43	126.66	510.59	0.0
max	1219.00	6.86	2887.13	326.36	1237.14	0.0

Table 18: GetSubAttributes: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.00
mean	1341.56	6.71	174.05	22.16	30.69	0.03
std	2.67	0.28	178.02	1.45	3.00	0.18
min	1336.00	6.29	46.32	20.38	27.34	0.00
max	1345.00	7.67	831.71	26.59	39.50	1.00

Table 19: UpdateAttributeValues: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.00
mean	1340.33	6.72	252.96	23.12	31.12	0.03
std	3.13	0.29	282.20	2.44	2.22	0.18
min	1334.00	6.14	50.83	21.03	28.68	0.00
max	1343.00	7.56	935.93	29.08	38.11	1.00

Table 20: UpdateAttributeValues: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1341.50	6.74	160.99	22.19	30.39	0.0
std	2.30	0.28	170.41	1.64	2.33	0.2
min	1334.00	6.22	49.53	20.63	27.72	0.0
max	1344.00	7.24	799.37	28.40	37.52	1.0

Table 21: UpdateAttributeValues: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1361.23	2.55	15.21	6.76	9.58	0.0
std	0.43	0.11	2.06	0.23	0.31	0.0
min	1361.00	2.28	10.97	6.36	9.04	0.0
max	1362.00	2.72	19.61	7.12	10.00	0.0

Table 22: GetCategoryMapping: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1360.23	2.38	14.58	7.34	10.49	0.0
std	0.77	0.17	1.70	0.23	0.30	0.0
min	1359.00	2.04	12.45	6.79	9.90	0.0
max	1361.00	2.63	18.64	7.67	10.99	0.0

Table 23: GetCategoryMapping: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1360.40	2.44	15.86	7.26	10.42	0.0
std	0.85	0.22	1.46	0.31	0.37	0.0
min	1359.00	2.03	13.11	6.54	9.76	0.0
max	1362.00	2.80	17.93	7.62	10.89	0.0

Table 24: GetCategoryMapping: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1303.23	2.24	970.60	53.40	210.76	0.0
std	36.51	0.17	736.84	31.77	218.35	0.0
min	1172.00	2.01	278.31	33.12	99.14	0.0
max	1327.00	2.70	2618.11	170.79	1001.95	0.0

Table 25: GetChildrenOrTopLevel: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1333.30	2.44	269.56	29.08	88.17	0.0
std	1.93	0.11	106.48	1.47	3.44	0.0
min	1329.00	2.19	144.27	26.05	81.56	0.0
max	1336.00	2.64	578.36	32.19	96.42	0.0

Table 26: GetChildrenOrTopLevel: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1333.23	2.47	280.86	29.08	87.22	0.0
std	1.79	0.12	138.06	1.38	3.97	0.0
min	1328.00	2.27	143.74	26.82	80.46	0.0
max	1335.00	2.75	819.39	33.30	95.66	0.0

Table 27: GetChildrenOrTopLevel: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1352.80	2.77	187.80	14.37	30.27	0.0
std	1.39	0.14	506.05	0.46	0.47	0.0
min	1350.00	2.58	37.27	13.73	29.29	0.0
max	1355.00	3.05	2857.09	15.89	31.82	0.0

Table 28: SearchCategories: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1325.46	2.55	2195.91	35.34	86.85	0.0
std	14.62	0.14	1371.23	11.52	46.55	0.0
min	1286.00	2.31	578.37	20.48	45.12	0.0
max	1346.00	2.90	6790.71	67.46	242.97	0.0

Table 29: SearchCategories: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1324.13	2.54	2276.06	36.19	92.98	0.0
std	16.55	0.29	1512.53	13.54	63.17	0.0
min	1272.00	1.21	657.47	19.36	45.45	0.0
max	1345.00	3.00	6550.59	79.76	289.26	0.0

Table 30: SearchCategories: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1357.76	4.07	19.31	9.28	12.98	0.0
std	0.43	0.20	1.94	0.27	0.35	0.0
min	1357.00	3.67	14.52	8.74	12.07	0.0
max	1358.00	4.42	22.84	9.82	13.50	0.0

Table 31: UpdateCategoryMapping: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.00
mean	1357.73	3.93	22.61	9.38	13.15	0.03
std	0.44	0.24	7.11	0.25	0.30	0.18
min	1357.00	3.43	16.25	8.92	12.55	0.00
max	1358.00	4.30	59.14	9.73	13.59	1.00

Table 32: UpdateCategoryMapping: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.00
mean	1357.76	4.09	24.09	9.30	13.05	0.03
std	0.43	0.25	6.46	0.28	0.31	0.18
min	1357.00	3.67	19.24	8.80	12.49	0.00
max	1358.00	4.50	55.35	9.75	13.51	1.00

Table 33: UpdateCategoryMapping: Descriptive statistics (CqrsApiMediatr)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1365.86	1.07	10.24	4.04	5.54	0.0
std	0.34	0.19	2.19	0.13	0.08	0.0
min	1365.00	0.07	6.55	3.68	5.34	0.0
max	1366.00	1.21	13.19	4.27	5.77	0.0

Table 34: GetRootCategories: Descriptive statistics (TraditionalApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1365.90	1.08	10.74	4.02	5.52	0.0
std	0.30	0.09	2.05	0.11	0.06	0.0
min	1365.00	0.93	6.97	3.82	5.40	0.0
max	1366.00	1.29	13.98	4.18	5.70	0.0

Table 35: GetRootCategories: Descriptive statistics (CqrsApi)

	reqs_count	req_dur_min_ms	req_dur_max_ms	req_dur_avg_ms	req_dur_p_95_ms	req_failed
count	30.00	30.00	30.00	30.00	30.00	30.0
mean	1365.96	1.09	10.98	3.94	5.44	0.0
std	0.76	0.09	2.27	0.39	0.48	0.0
min	1365.00	0.87	5.72	2.42	2.92	0.0
max	1368.00	1.23	14.40	4.25	5.69	0.0

Table 36: GetRootCategories: Descriptive statistics (CqrsApiMediatr)

B Figures

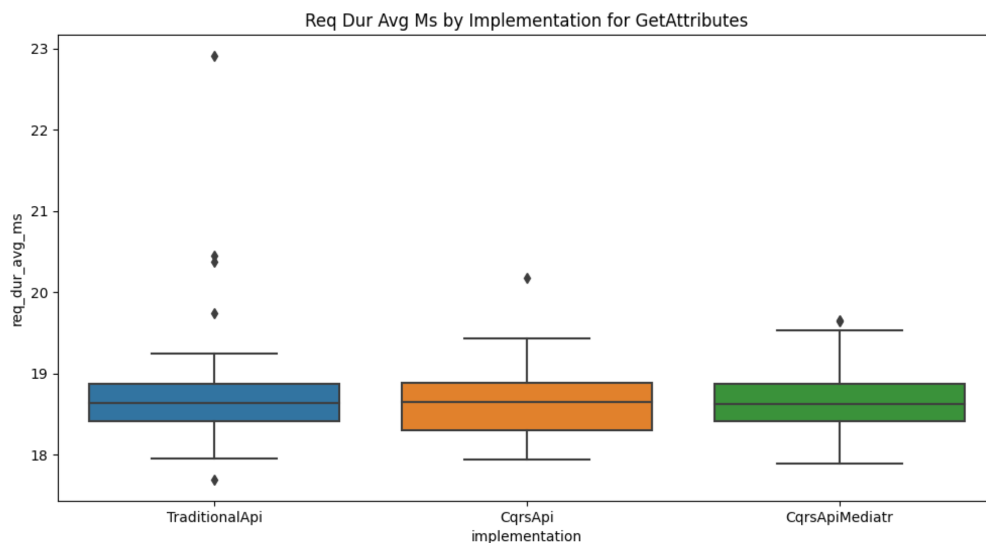

Figure 4: GetAttributes average in milliseconds

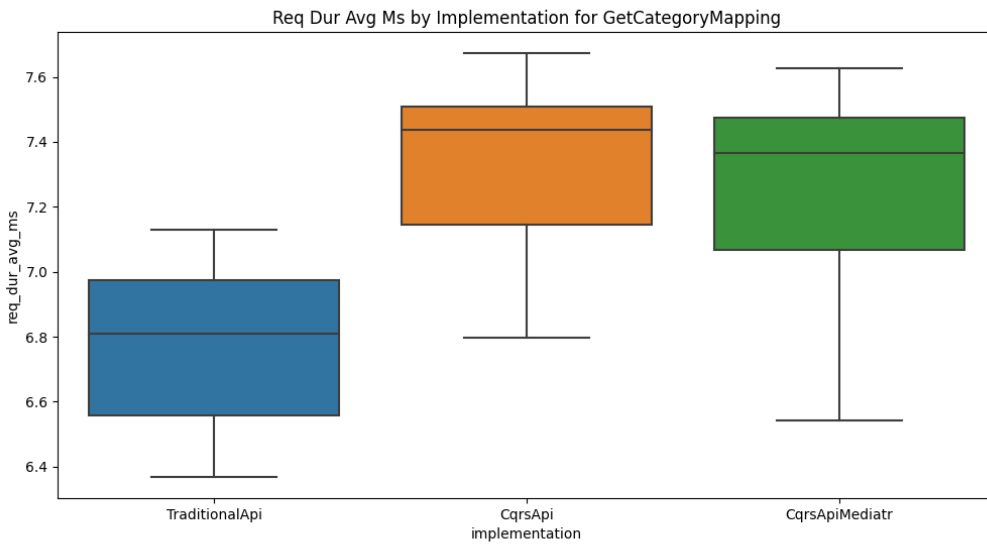

Figure 5: GetCategoryMapping average in milliseconds

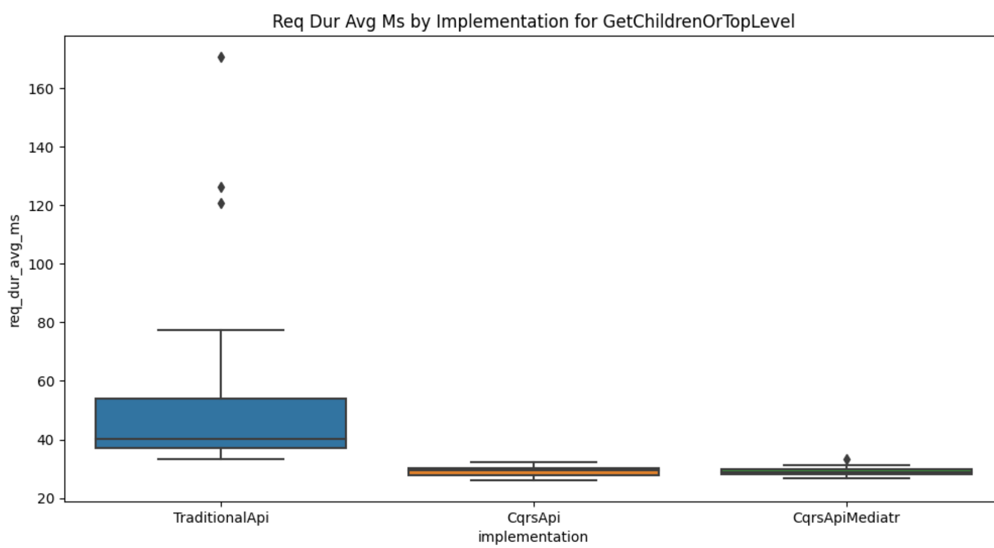

Figure 6: GetChildrenOrTopLevel average in milliseconds

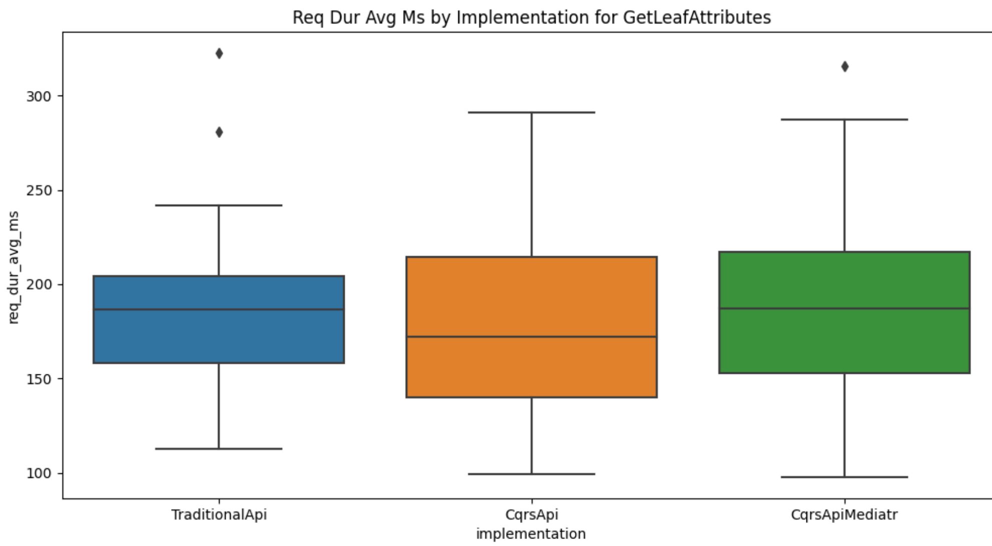

Figure 7: GetLeafAttributes average in milliseconds

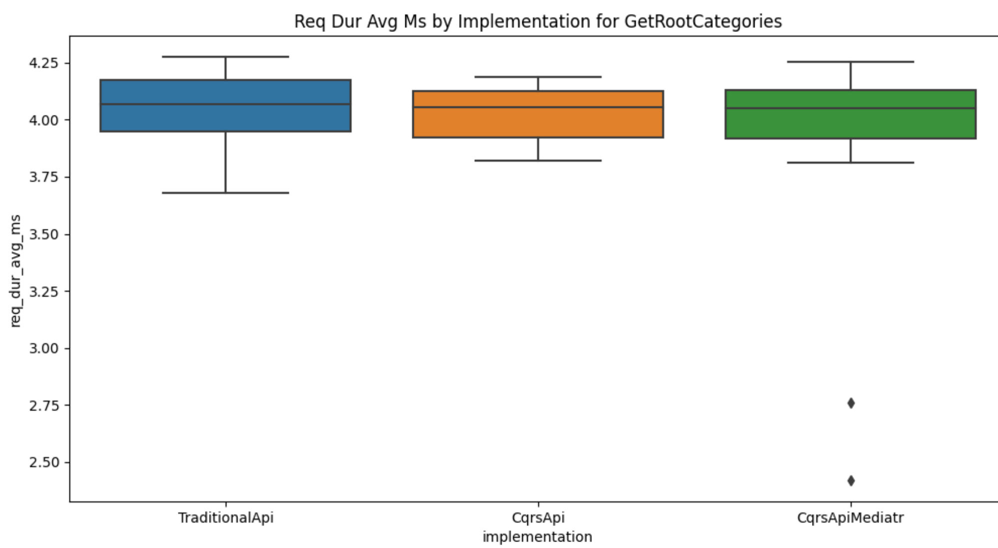

Figure 8: GetRootCategories average in milliseconds

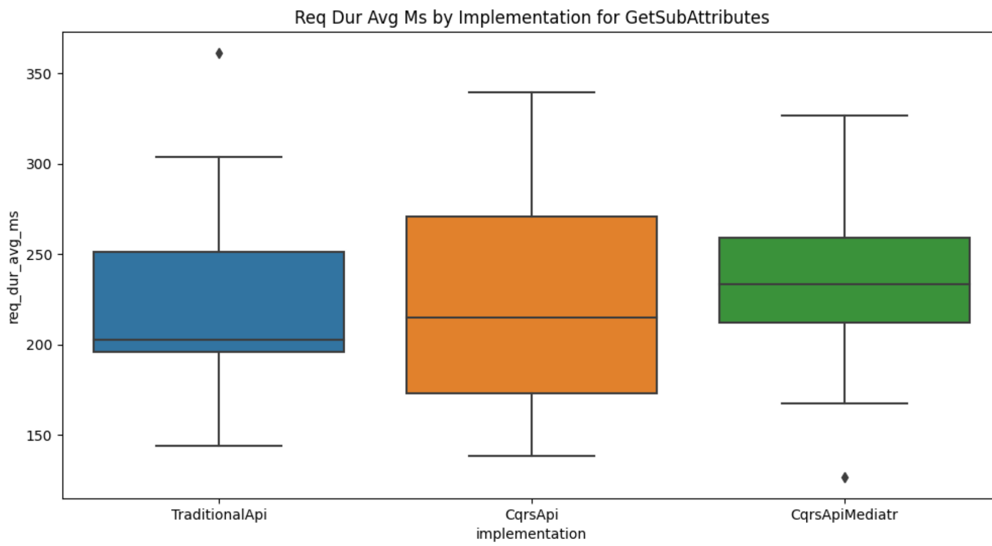

Figure 9: GetSubAttributes average in milliseconds

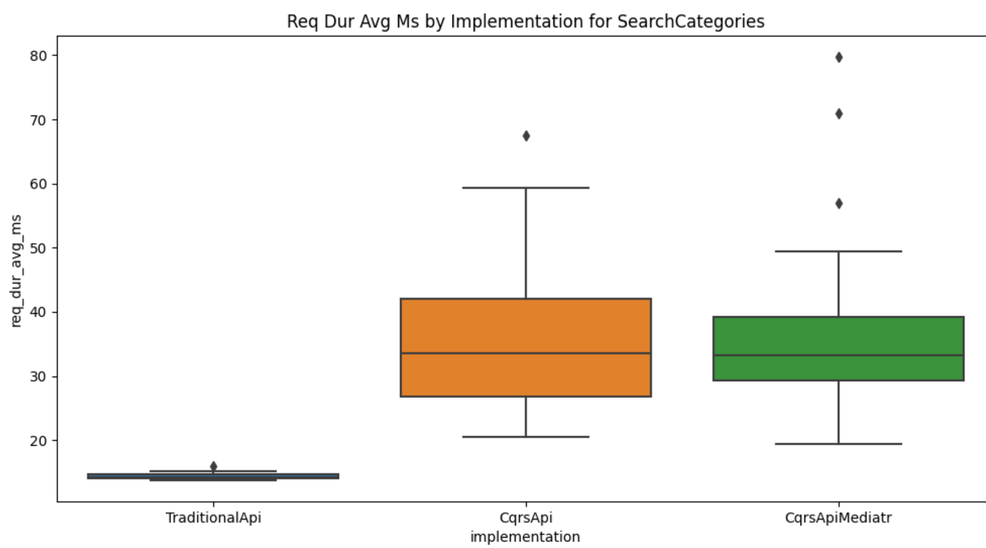

Figure 10: SearchCategories average in milliseconds

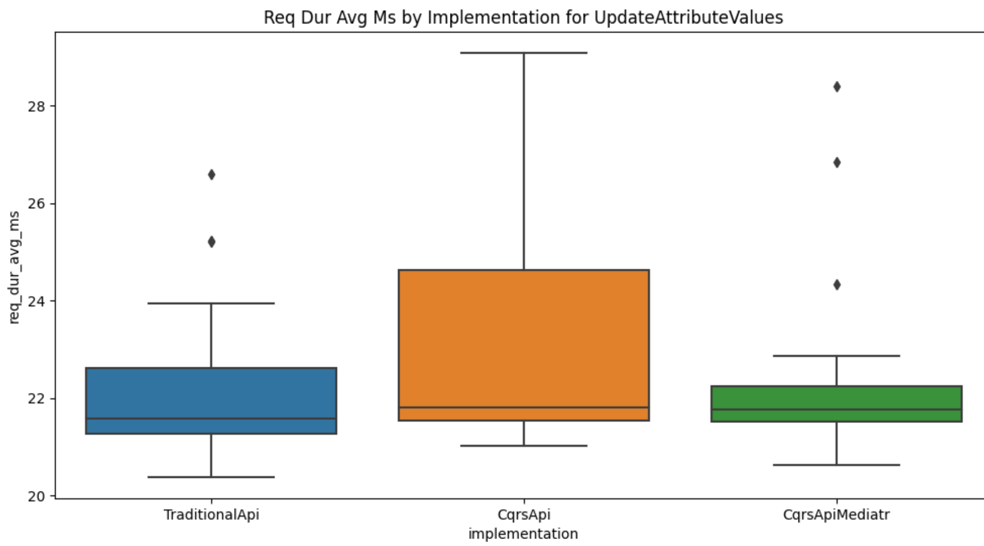

Figure 11: UpdateAttributeValues average in milliseconds

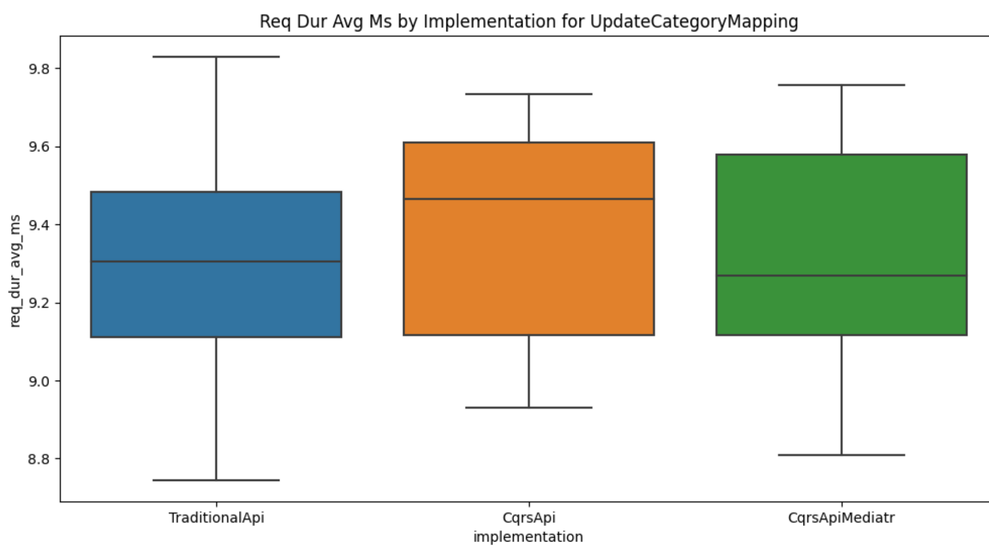

Figure 12: UpdateCategoryMapping average in milliseconds

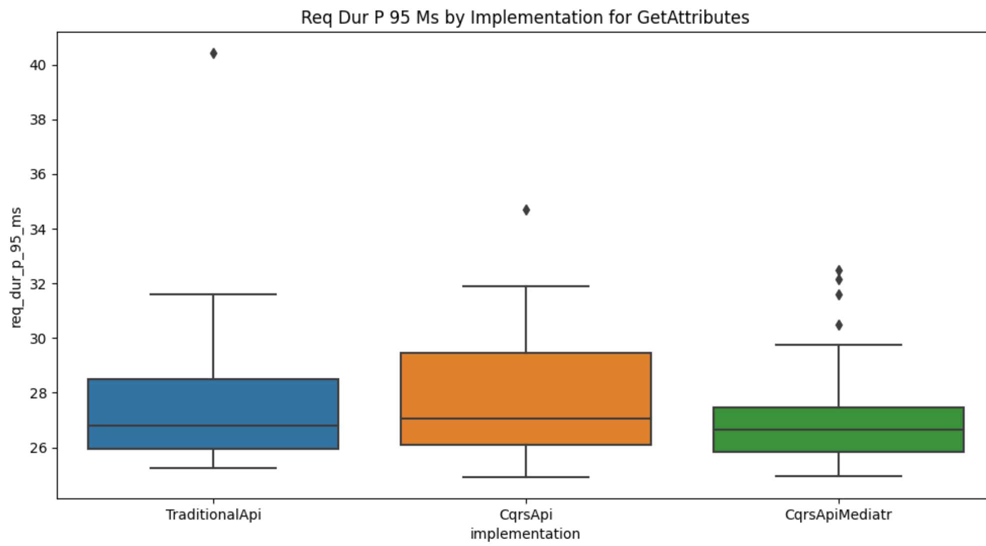

Figure 13: GetAttributes 95th Percentile in milliseconds

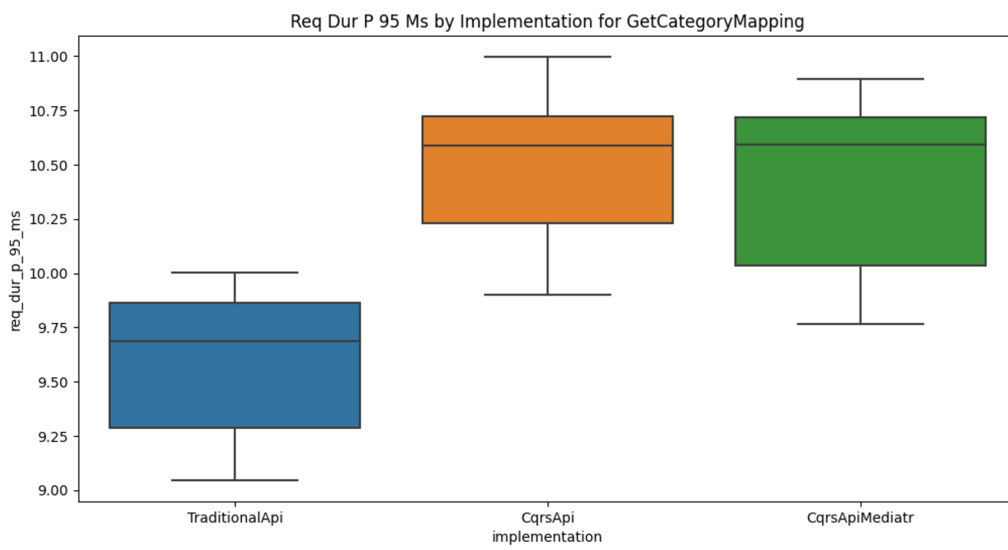

Figure 14: GetCategoryMapping 95th Percentile in milliseconds

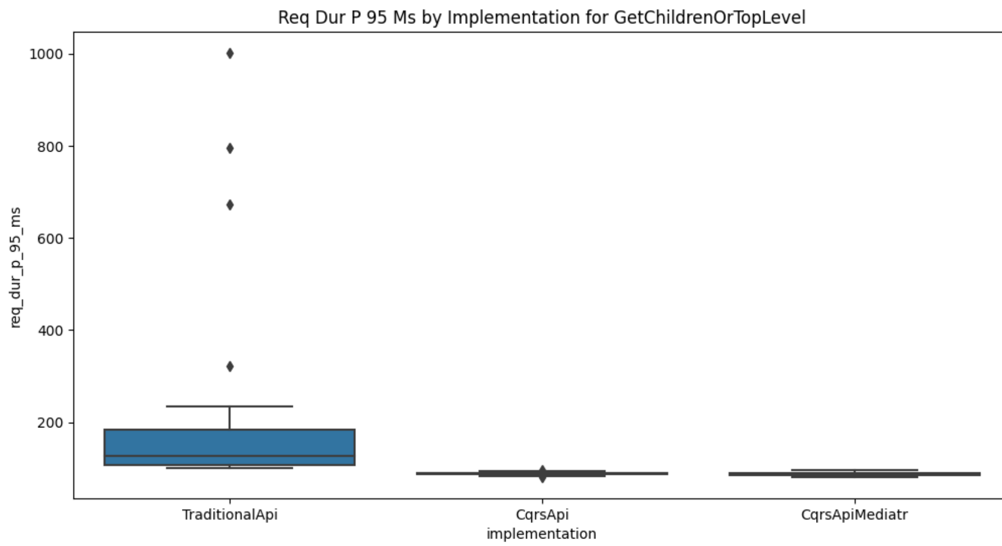

Figure 15: GetChildrenOrTopLevel 95th Percentile in milliseconds

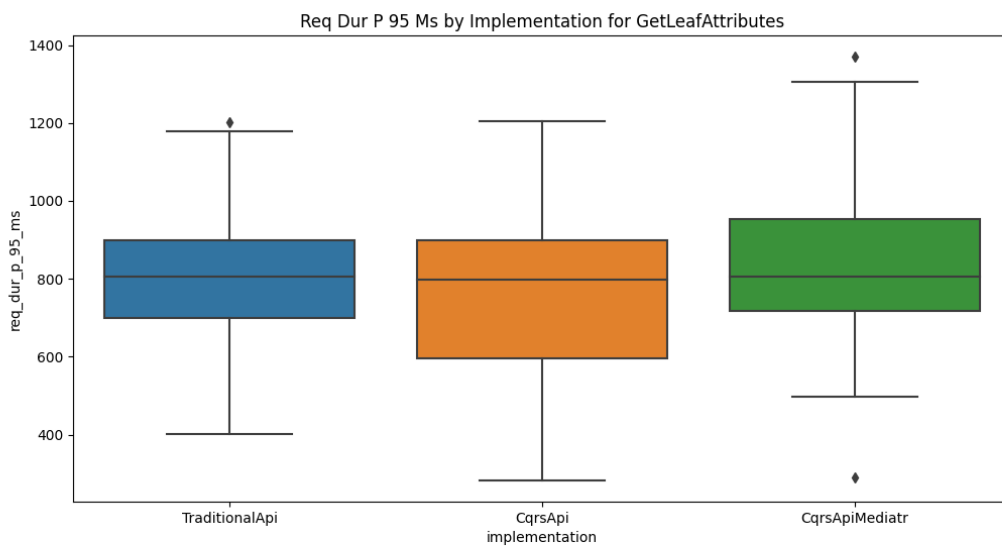

Figure 16: GetLeafAttributes 95th Percentile in milliseconds

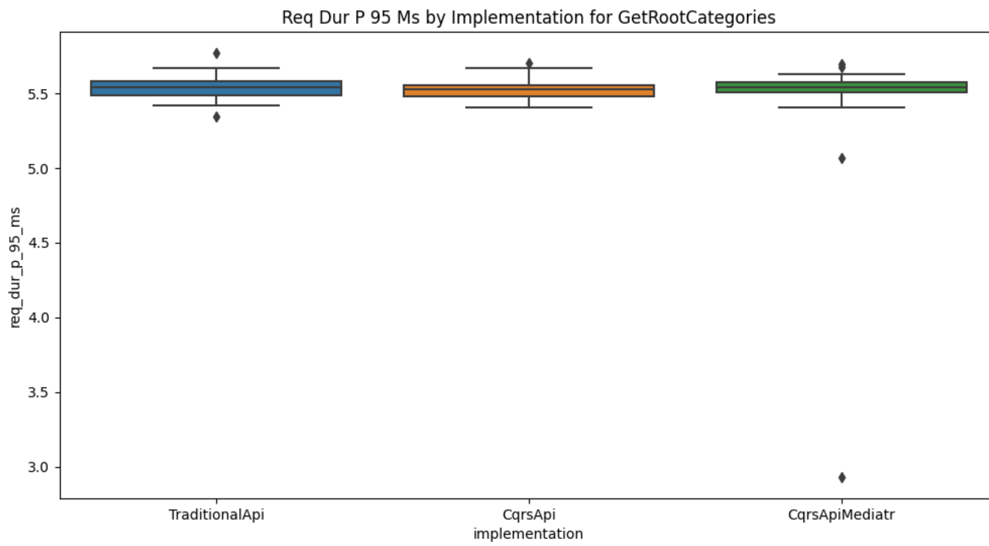

Figure 17: GetRootCategories 95th Percentile in milliseconds

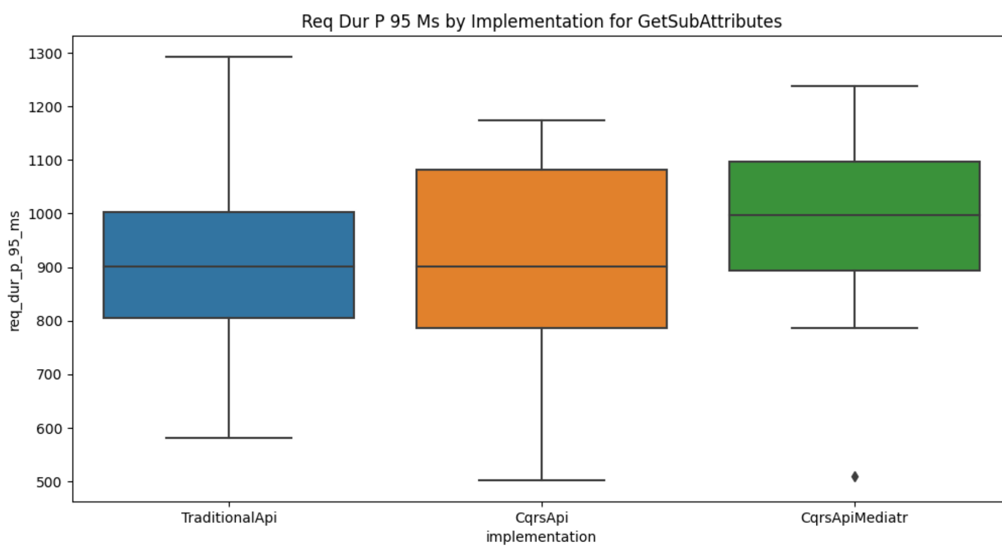

Figure 18: GetSubAttributes 95th Percentile in milliseconds

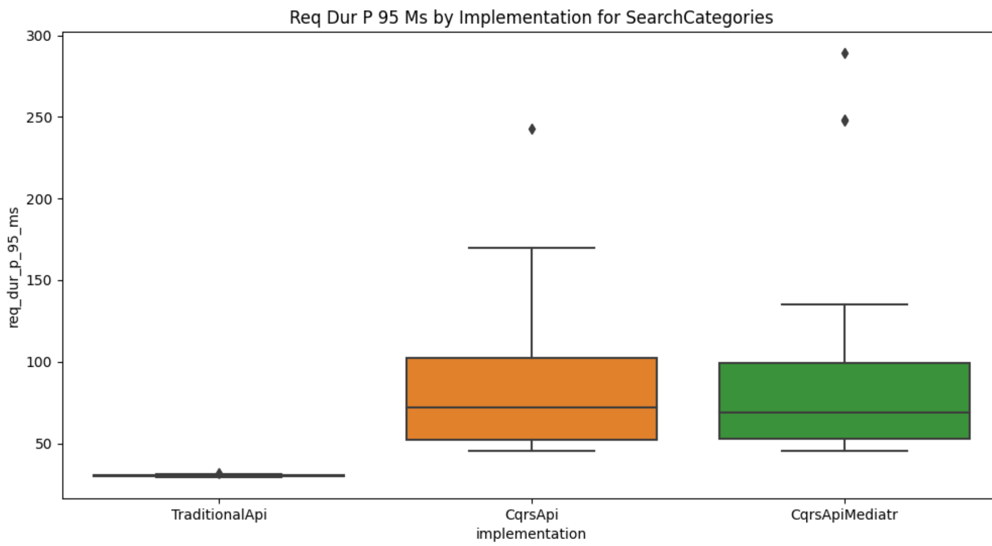

Figure 19: SearchCategories 95th Percentile in milliseconds

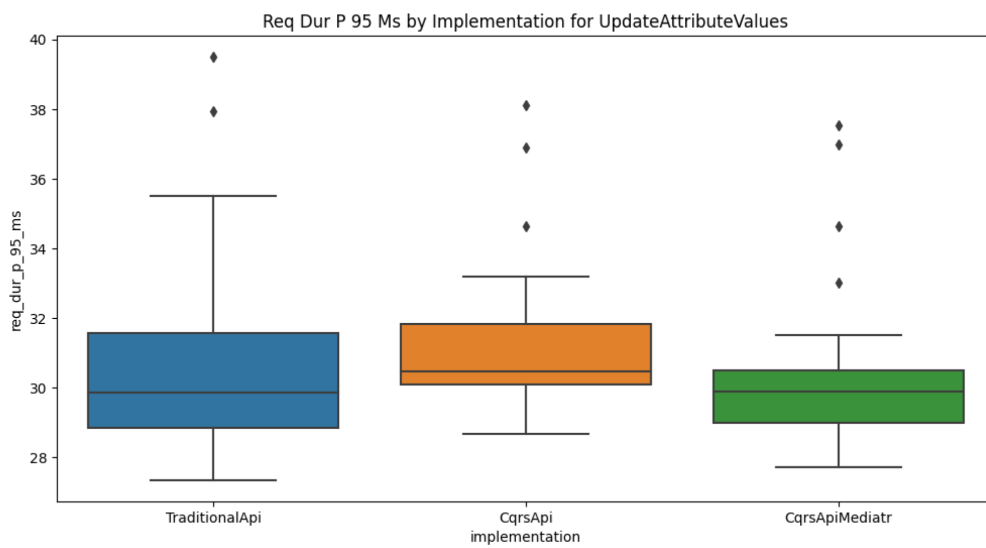

Figure 20: UpdateAttributeValues 95th Percentile in milliseconds

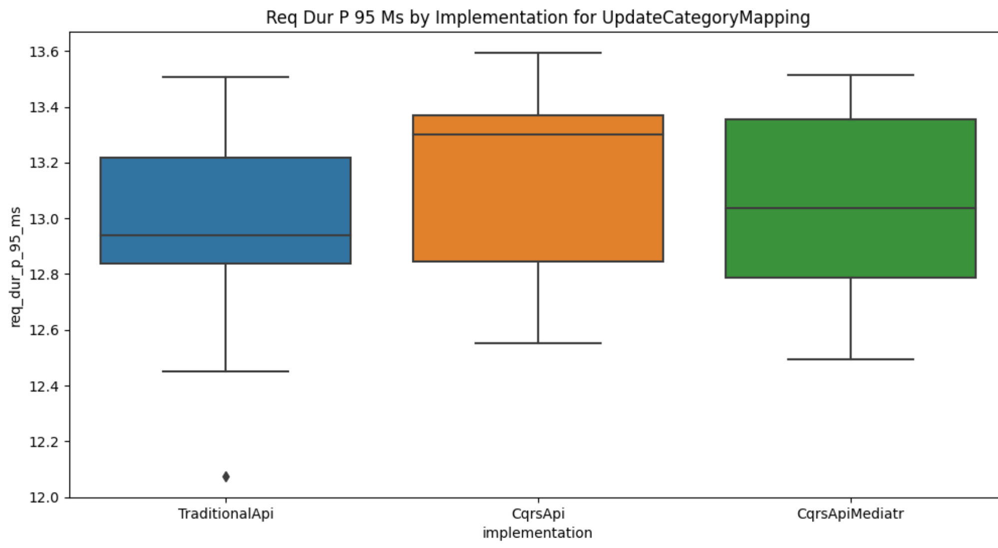

Figure 21: UpdateCategoryMapping 95th Percentile in milliseconds