BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Performance optimization in the context of highly scalable global APIs

Supervisor
Lect. Dr. Mihai Andrei

Author
Andrei Moldovan

2025

ABSTRACT

As time goes by, the world becomes more digitalized and inter-connected, and more and more businesses move their operations online. The user base for many applications is constantly growing, along with the expectations of the users. In-demand applications with high traffic are getting thousands of requests per second, if not more, from every corner of the world. In today's ever-evolving digital landscape, users expect fast, reliable apps, which are always available and provide the required information, with a focus on speedy delivery. Any application with the target of producing value and reaching as many users as possible, should be architected and implemented from the start with the above-mentioned characteristics in mind.

This thesis aims to analyze the aspects that need to be taken into consideration when developing a fast and reliable application meant for global usage. The focus will be placed on the architectural patterns and performance optimization techniques that can be used to develop, deploy and test an efficient, resilient and highly scalable global API.

The thesis will take a look at different architectural patterns, discussing the advantages and disadvantages, along with best practices, recommended approaches and common pitfalls. Next it will delve into database design best practices and optimization techniques. Subsequently, the tools and technologies used for implementing the application will be explained. Afterwards, two different implementations of the same API will be presented and load and stress tested. Finally, based on the test results, their respective performance will be compared, with regards to resource consumption, request throughput and response time, which will highlight the effect of the previously discussed architectural patterns and performance optimization techniques.

Contents

Chapter 1

Introduction

In the past years, a huge increase in the capabilities and offerings of cloud providers could be seen, along with an increase in cloud adoption by businesses and IT professionals. This widespread migration of applications in the cloud happened because both the needs of businesses and the expectations of users have evolved, concepts like high availability, scalability, resilience, disaster recovery and high performance becoming paramount to all user-facing applications. This was facilitated, or made use of architectural patterns such as microservices and event-driven architectures, and technologies such as containerization, orchestration, etc., in contrast with older more traditional architectures such as monoliths.

The purpose of this paper is to present and discuss the differences, the pros and the cons between these different approaches and architectures. Then it should highlight the techniques and patterns that can be used in today's technological landscape to develop an API that can be considered highly efficient, especially with respect to scalability and performance. This will be achieved by developing the same API with two different implementations, one using microservices and the other one being a classical monolith. These two implementations will be functionally identical, will work with exactly the same data and will be subjected to the same virtual load, using load and stress tests. This load should be realistic and as close as possible to real traffic conditions and user behavior. The test results and the resulting metrics and statistics will be analyzed and compared. Finally, this analysis aspires to provide a comprehensive explanation as to what makes an API efficient, how such an API can be developed and deployed, and why this rise in cloud-native and distributed applications is happening, and the benefits it brings with it.

The second chapter of this thesis will present in detail the differences between microservices and monolith architecture. It will discuss the advantages and disadvantages of each of them, and present some common best practices and design patterns to keep in mind when developing the code and the architecture of cloud-native applications.

The third chapter will go into some detail with regards to database design and optimizations at the database level, namely how indexes should be used and created, and how queries should be built around them. Then it will delve into common pitfalls that may happen when using ORM technologies and how these can be avoided, and how the performance of queries generated using ORM can be improved. Afterwards, it will provide an explanation as to why these aspects are important from the early life-cycles of any application.

The fourth chapter should explain what technologies have been used for developing the application, and why these technologies have been chosen. The benefits these tools bring to applications running in a distributed cloud environment will be highlighted.

The fifth chapter will dive into the particulars of the actual application. It will explain what this API should do and why and how the business features of the API were chosen. It will present the process of collecting and normalizing a realistic dataset, meant to be used for the database of the application. Then it will go into detail with regards to the whole architecture of the app, how traffic is routed and controlled, how internal firewall rules are enforced, how the microservices are built and deployed, how the microservices communicate with each other and the internet, how the application is secured, how load balancing, service discovery and auto-scaling work in the context of this application, and the reasons behind all the architectural choices. Subsequently, a user guide will be presented which will explain how the API can be used, and present some typical user journeys. Eventually, the load testing configuration and scenarios will be discussed. This section will present the test results and explain how and why each of the API implementations behaves differently. Ultimately, the reasons behind the performance or lack thereof of each implementation will be detailed.

Chapter 2

Microservices vs Monolith Architecture

The main paradigms of software architecture are monoliths, microservices, and serverless, although the latter could be considered a more specialized form of the microservices approach rather than its own thing.

These approaches have been heavily scrutinized and discussed in recent years. Each of them has its own pros and cons, and developers must be familiar with all of them and their distinctions in order to properly build efficient and scalable applications, with regards to the business requirements and their use cases.

2.1 Monolith

This is probably one of the most widely known and used design patterns when it comes to software architecture, especially in the context of enterprise applications. It's main characteristic is the fact that the whole application is built, packaged, and deployed as a single unit. It can be deployed on one or multiple machines, on premise or in the cloud, but it is always a single executable that runs the whole application.

One can imagine that since it is only one unit of development, getting started on a project with such an architecture is fairly easy. It is usually built and managed by one team, which speeds things up since it does not necessarily require communication with other departments. There is only one code base, one pipeline, one deployment mode, and one database for developers to worry about. Even in the case of a distributed database, which is usually achieved through replication, be it logical or physical, all operations on the database can still be performed through transactions that can be easily controlled and rolled back by the application, ensuring data consistency at all times. Automated and integration testing can also be

done without many issues, since it is only one application with a given set of entry points that is being verified.

Taking this into account, one can see why this would be the initial choice of a team or company when starting a new project. But after some time, the code base inevitably grows to considerable proportions, because of the constant additions of new business features, requirements, and improvements. This is applicable for almost all applications. This is usually the point where problems start to arise, especially if the application is subject to an increasing number of users.

After this threshold, various problems start to occur. If the developers want to upgrade the version of some underlying dependencies or framework, the whole code base needs to be checked and updated, which can be quite time-consuming.

The more the application and its usage grow, the more resources it needs, hence the more resources its underlying machine needs. The problem is that always having to upgrade your servers to support the app is very costly and not sustainable, since it requires constant maintenance and deployments of the app, which can have a negative impact on the business. This is also usually quite wasteful, since most apps face some peak times when they're under heavy load and they actually need those extra resources, and a lot of times when they're not, they could be running with a lot less.

Issues in some parts of the application have the potential of becoming bottlenecks for the entire app. We can easily see how some requests which are not used that much, and which are not the focus of the business, if not properly implemented or optimized, have the potential of consuming a lot of resources and decreasing the overall performance of the app, sometimes even blocking other requests or users from performing their desired function.

The more the app grows, the more the issues start mounting, until the negative effects on the business can no longer be ignored. The following table taken from [AK23] highlights the mentioned problems, albeit at a small scale (2.1).

Test scenario	Response time	Throughput
Single user load	200ms	100 req/s
100 Concurrent Users	2s	50 req/s
1000 Concurrent Users	20s	10 req/s

Table 2.1: Monolith performance benchmarking results

A monolith application request flow might look like this: 2.1.

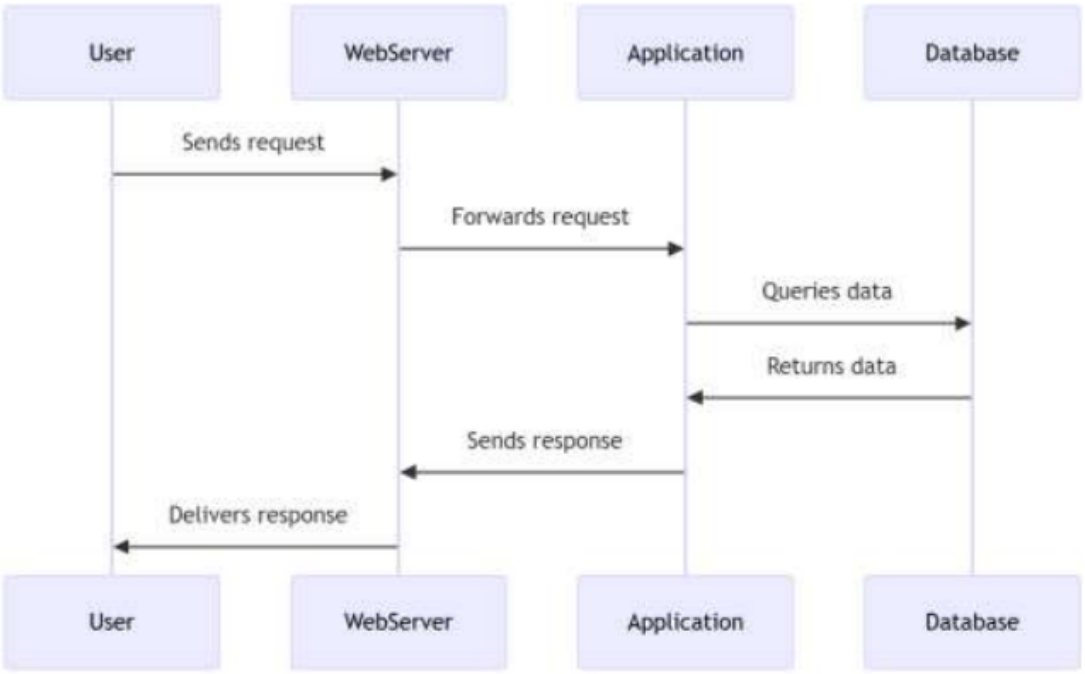

Figure 2.1: Monolith request flow [AK23]

2.2 Microservices

This architecture paradigm is a bit more recent compared to monoliths and was created to address the ever-evolving needs of enterprise applications and the issues brought on by monolithic architecture. The rise of different cloud technologies and providers, and the launch of various containerization and orchestration technologies helped microservices gain popularity and increased interest from the programming community.

The main difference with this approach is that the application is now modularized, each module, also called microservice, being its own isolated app serving a very specific business purpose. This effectively turns a huge single application into an ecosystem of multiple small applications, all of them communicating with each other in this ecosystem.

The main advantage of this is the resource allocation and horizontal scaling. Unlike monolithic applications, where scaling requires replicating the entire application, microservices enable organizations to scale only the services that experience increased demand [OCO24]. Instead of allocating more resources to the same application and server while the app grows, we can fine-tune and tailor the resources specifically to those parts of the application that need them. Then, if a specific part of the app requires more resources, we can simply create more instances of the corresponding microservice to accommodate this load. We can also introduce autoscaling technologies, so that those extra instances are created only when needed, for

example during peak times, and then removed during periods of less traffic. Also, we don't really have to care where these microservices run, as long as they are able to communicate with each other. They can run as containers on one machine, as different apps on different virtual machines or as anything in between. This distributed nature makes them perfectly suitable for a cloud environment, where infrastructure is more-or-less abstracted to the developers.

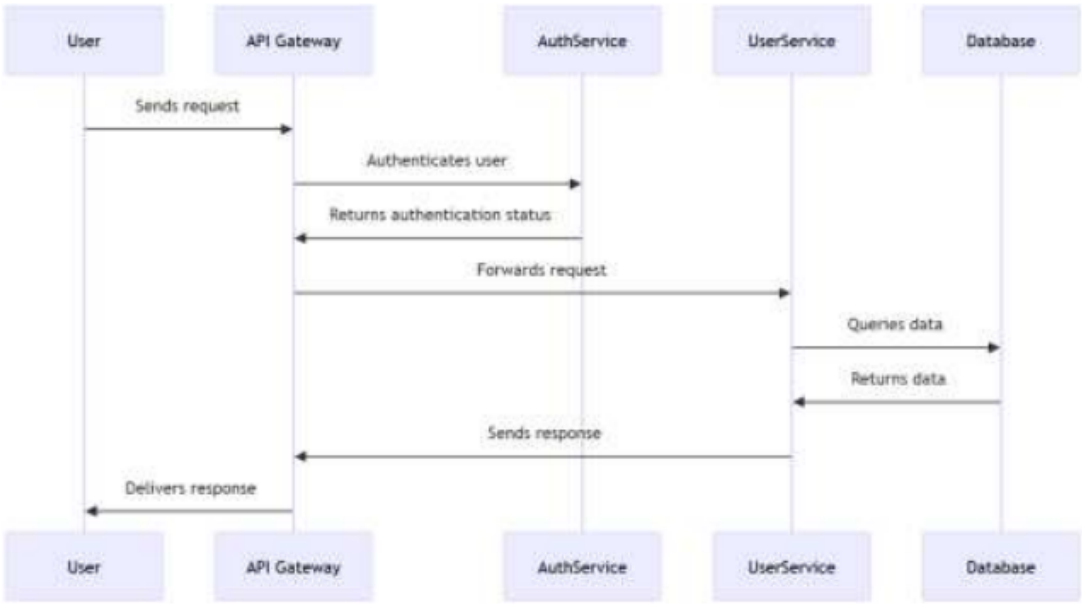

Figure 2.2: Microservices request flow [AK23]

Besides resource consumption optimizations, microservices are also easier to maintain, build and deploy. Fixes, updates or improvements can be performed in a single or a couple of microservices, exactly where they are needed, without having to go through the whole app. These in turn can be built, tested and deployed in a very small time frame, without affecting the overall app and usually without noticeable down-times for the user.

Another advantage is that we no longer have to worry about bottlenecks in the app, since if one service fails, it does not necessarily bring or slow down the entire application [OCO24]. Depending on how fault-tolerance and automatic recovery are implemented, traffic can be re-routed to other fail-over instances, which can pick up traffic temporarily, or a circuit-breaker can stop traffic for a while, to give the affected service enough time to recover on its own. Also, the overall system fault tolerance increases when services distribute their copies across multiple worldwide locations [Tho20], since traffic can be re-routed to servers running in other geographical locations, in case of physical issues such as hardware problems or disasters.

But this distributed nature of microservices brings its own difficulties and challenges. The architecture becomes much more complicated which requires much

more specialized teams. Also, multiple teams might work on multiple microservices within the same application, which introduces challenges of inter-departmental communication. The app becomes heavily reliant on network conditions and architecture, since multiple microservices are communicating with each other. Since usually there are multiple databases involved in a microservice ecosystem, transaction handling becomes much more complicated. A logical transaction can end up being split between multiple services and multiple database transactions, which complicates rollbacks and data consistency. Overall, this causes the initial learning curve and implementation time of a microservice based app to be higher when compared to a monolithic app, but then it smoothens out later in the life of the application.

As far as performance goes, microservices usually handle load and peak times a lot better than monoliths, as illustrated here [AK23]: 2.2

Test scenario	Response time	Throughput
Single user load	180ms	110 req/s
100 Concurrent Users	1.8s	15 req/s
1000 Concurrent Users	15s	12 req/s

Table 2.2: Microservices performance benchmarking results

All of these pros and cons need to be considered when choosing an architectural design for the application, because further issues which might occur and further optimizations which might be performed are strictly dependent on how well the architecture fits the use cases of the business.

2.3 Cloud native mindset

Implementing an application using microservices requires a so-called cloud native mindset. This implies that when developing the application, there are more things to take into consideration besides the actual application itself. Thought needs to be given to how the microservices will communicate with each other and how they will exist in a cloud environment.

In order to take full advantage of the distributed nature of microservices, implementing auto-scaling is recommended. This ensures that the services always have exactly the needed amount of resources and capacity to serve their incoming traffic, and they're always available to serve requests. Auto-scaling also requires some form of load balancing, because incoming traffic should have an even distribution across all the running instances of the services. This ensures that all instances are put to work, and situations where an instance is sitting idle, while another is overloaded, are avoided. Besides all this, some form of routing or API Gateway, to filter

all incoming requests and forward them to the appropriate service, is required.

Depending on the system and the business requirements, the services can also communicate between themselves. In such a case, the communication can happen either in a straight-forward blocking manner, with HTTP calls or gRPC, or in an asynchronous non-blocking manner, using messaging technologies and protocols, such as Kafka or AWS SQS. Each of these implementations is valid and highly dependent on what the system is supposed to do and how it is supposed to behave, and each of them brings with it further considerations.

In the case of blocking communication, some form of service-discovery is usually required, in order to enable dynamic service resolution and decouple service interactions from specific network locations (Gilbert, 2018, as cited in [OCO24]). Distributed transactions also have to be taken into account here, since a logical operation can end up being split between multiple services and databases. This can be implemented through a series of sequential database transactions, which are then orchestrated to ensure that the final state of data across all of the databases is consistent, otherwise known as the SAGA pattern.

In the case of non-blocking communication, other models may be preferred, such as the BASE transaction model, which follows slightly different principles as compared to ACID, and is more suitable for a distributed cloud environment. This ensures eventual data consistency, so there may be moments where data is not fully consistent across the system, but eventually, after all events are processed, it will be.

Last but not least, fault-tolerance and fail-overs should be taken into consideration. What happens if a service is unavailable for some time needs to be analyzed. Will the other services be affected? Should traffic going to that service be redirected to some other service instead? Should requests going to that service be paused for some time to give it time to auto-recover?

Developing a microservices application with these aspects in mind helps ensure a robust cloud-native design, which will determine if the application runs efficiently in a cloud environment or not, and if it is able to scale to accommodate high loads.

Chapter 3

Tools and technologies

Microservices are meant to be used and are best-suited for running in a distributed cloud environment. In order for the microservices implementation of the API this thesis focuses on, to be well-architected and efficient, the use of corresponding cloud-native tools and technologies was required. Most of them belong to the cloud-native landscape, as defined by the CNCF (Cloud Native Computing Foundation) [CNC]. This chapter will go on to explain the purpose and reasoning behind the chosen technologies.

3.1 Java

Java is one of the most well-known programming languages, GitHub listing it as the third most popular programming language on the platform in 2022 [Top]. It is a high-level programming language, focused on object-oriented programming, taking its inspiration from other languages such as C++, but extending upon it. Some of the aspects that helped it gain wide usage and popularity are its ease of use, its automatic memory management using garbage collection technologies and the fact that it's able to run on any device, since it's compiled to bytecode which runs on a JVM. This also makes the development of Java applications system-independent, which is another benefit for the programmers. It is general-purpose and very robust, many of its different editions providing comprehensive support for all sorts of application development, such as enterprise applications or mobile applications, being suited for both front-end and back-end development. Other notable features of Java are its extensive support for multi-threading and its Just-in-Time compilation, combining the advantages of traditional compilation with the advantages of an interpreter, for speed and flexibility at run-time.

The APIs of this application were implemented with Java, mainly because of its ease-of-use and robustness.

3.2 Quarkus

Quarkus is a modern Java framework, specifically implemented and tailored to be used in applications running in cloud environments. What makes Quarkus stand out when compared to other more popular Java frameworks such as Spring Boot, is that it's heavily focused on performance. Unlike other frameworks, a lot of the work related to executing Java applications is moved from the run-time phase to the build phase. During the build phase, Quarkus reads the application configuration, scans the classpath for annotated classes, removes unnecessary classes or methods, and constructs a model of the application [QUA], containing exactly the instructions required during startup. By doing this, the build-phase generates highly optimized code, which leads to much faster startup times and better JIT optimizations during run-time. This ensures that the overall app is much faster compared to apps developed using other frameworks. The memory footprint of Quarkus applications is also much smaller when compared to Spring Boot, since instead of relying on reflection during run-time and dynamic proxies, the byte-code generated by the build phase already contains all the necessary information for running the app and uses regular invocations and custom proxies which were already generated.

Under the hood, Quarkus runs on a reactive engine, built on top of Eclipse Vert.x. So instead of using a traditional thread pool for handling requests, it employs a few event loop threads which further delegate work to some non-blocking I/O worker threads. So regardless of whether the code itself is written in a reactive or imperative manner, the execution is much faster since the I/O layer is highly optimized. This makes Quarkus really flexible, since it's very well suited for implementing classic imperative REST APIs, reactive REST APIs, event-driven architectures and much more.

As far as development goes, Quarkus is really easy to use and speeds up the overall development process. What enables it to do this is a live-reload functionality, a single configuration file which can be used to serve multiple profiles, and the fact that it's less verbose than other frameworks.

All microservices of the application were developed using Quarkus, most of them using an imperative programming style, and one of them fully taking advantage of the reactive capabilities of Quarkus. The custom API Gateway of the application uses only non-blocking I/O threads to process and forward requests to the other services. This will be detailed and explained further in the following chapters.

3.3 Keycloak

Keycloak is an open-source authentication and authorization server, which is highly configurable to fit everyone's needs. It's built with Java, hence it can also be easily extended with custom Java code by extending different SPIs (Service Provider Interfaces). It provides already built docker images, but those can also be extended to create custom-built highly optimized docker images meant for production environments. This can be achieved only by changing some configuration parameters and building the artifacts required by the application in a separate docker build phase, before the final image is built. This makes the final images really lightweight and efficient.

It provides support for all traditional features of authentication and authorization servers, such as single sign-on, OIDC and OAuth 2.0 support, LDAP support, role based management, etc.

For this application, Keycloak was used as an authentication and authorization microservice, for creating and managing API clients, together with role based management based on the clients' scope. The scopes were relevant for rate-limiting the API clients based on different criteria.

3.4 PostgreSQL

PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned a strong reputation for reliability, feature robustness, and performance [Pos]. It is easy to use, highly efficient and flexible, providing support for multiple schema and multiple types of columns and indexes, such as JSON data types, ARRAY data types, specialized GIN indexes meant specifically for composite values such as the ones stored by the previously mentioned JSON and ARRAY data types, etc.

It also integrates well with a lot of other systems, such as AWS, which provides built-in support for configuring a lot of different PostgreSQL databases with its RDS service. The reason PostgreSQL was used for this thesis, was the Aurora Serverless database type provided by AWS, which allows a PostgreSQL instance to auto-scale in a given pre-configured capacity interval, without manual intervention.

3.5 AWS

AWS is one of the most well-known and used cloud providers in the world. It basically provides support for creating and managing any kind of infrastructure

one might need in the cloud, such as regions, availability zones, virtual private networks, firewalls, routing, DNS, load balancing, etc. It also integrates well with IaC tools such as Terraform.

As far as application development goes, it provides support for hosting private docker image repositories, launching and auto-scaling containers automatically on demand, logging and monitoring for applications, hosting databases, etc.

AWS was used for this thesis for hosting and deploying the whole application, and for creating an efficient cloud architecture in order to be able to perform relevant load-testing scenarios.

3.6 Artillery

Artillery is a load testing platform, providing a CLI tool for executing tests, which can then create different dashboards and statistics on their cloud platform, for further analysis. Artillery enables the user to define multiple test scenarios and configurations to simulate real user behavior and real load on the application. This is achieved by specifying how many virtual users should be created each second, in different phases, and what actions these users should perform on the system under test.

Artillery integrates well with AWS, providing support for launching multiple containers responsible for spawning the virtual users that will test the application. This is ideal for load testing with a big number of users and requests that run in parallel, just like in a real life scenario, which can not really be achieved if the same tests are run on a local machine.

Artillery was used during the development of this thesis for load and stress testing the two implementations of the API, the microservices one and the monolith one. The tests were performed with different real-life scenarios in mind, and the analysis tools provided by Artillery's cloud platform helped with analyzing the test results, by comparing different metrics and statistics between the tests.

3.7 Angular

Angular is a front-end development framework, maintained by Google. It is an opinionated framework, providing a clear rule-set with regards to the structure of the application. Applications developed with Angular are composed of multiple components, which are then injected wherever they're needed using a flexible dependency injection mechanism. This effectively enables the developer to create a modularized, loosely-coupled application. Another notable feature which makes it

stand out across the JavaScript ecosystem is its use of Typescript, which enables the use of strict data types, when compared to the usual flexible approach to objects and types of JavaScript.

I have used Angular for developing the front-end for this thesis, mainly because of its component model and because of Typescript. For me it was more familiar than other front-end frameworks, since I found it more OOP leaning and closer in resemblance to back-end development technologies.

3.8 Postman

Postman is a platform which can be used for testing and verifying APIs. It enables the user to configure and send requests, using different communication protocols, authentication schemes and payloads. It can be used for testing HTTP and HTTPS endpoints, websocket communication, etc. I have used it extensively for the integration-testing of the APIs of this thesis.

Chapter 4

Application

The application itself is a REST API. The API has two implementations, one being a monolith, and the other an ecosystem of microservices, both hosted in the cloud with AWS. The main focus of this paper is to provide a performance comparison under heavy load between these two implementations, and to analyze and highlight the design choices and techniques that were used in order to increase the performance and scalability of this API. The comparison will be performed with regards to different metrics, such as response time, error count, throughput, resource consumption, etc.

4.1 Solution

The API is meant to be used by clients worldwide with a monthly tiered subscription model. The main business purpose of the API is to be used as a backend for Coffee E-Commerce Websites. E-Commerce Websites can get a lot of traffic, both read and write oriented, so this felt like a valid scenario for a discussion about performance optimization and scalability.

The subscription has two tiers, a basic one and a premium one. Each client is rate limited to a certain number of requests in a given time frame depending on the chosen subscription tier. These properties are configurable, but for our example here, the basic tier is limited to 50 requests per hour, and the premium tier to 100 requests per hour.

The authentication of the clients is performed with machine to machine credentials, using a custom Keycloak microservice as the authentication provider. Clients are created by an admin user through the API. When a client is created in the application, it also gets created in the authentication provider with the corresponding scope, depending on the chosen subscription. This scope is later used for authorization and rate limiting. The admin user then provides the generated client credentials

to the client.

To help with creating a higher load on the application I decided to use a fairly moderate dataset, with a high hierarchy between the tables and the entities implicitly. This helps with creating more complex search queries, which return bigger datasets. Overall this helps increase the load on the databases, and in turn also on the application. The data was extracted from a raw CSV file from a GitHub repository [Jam], which collected the data from the 2018 review pages of the International Coffee Quality Institute [CQI]. Initially the data looked like this: 4.1

	A	B	C	D	E	F	G	H
1	Country.of.Origin	Farm.Name	Mill	Company	Region	Numbe	Bag.We	Harvest.Year
2	Ethiopia	metad plc	metad plc	metad agricultural developmet plc	guji-hambela	300	60 kg	2014
3	Ethiopia	metad plc	metad plc	metad agricultural developmet plc	guji-hambela	300	60 kg	2014
4	Guatemala	san marcos barrancas			san marcos	5	1	
5	Ethiopia	yidnekachew dabessa coffee plantation	wolensu	yidnekachew debessa coffee plantation	oromia	320	60 kg	2014
6	Ethiopia	metad plc	metad plc	metad agricultural developmet plc	guji-hambela	300	60 kg	2014
7	Ethiopia	aolme	c.p.w.e		oromia	300	60 kg	
8	Ethiopia	aolme	c.p.w.e		oromiya	300	60 kg	
9	Ethiopia	tulla coffee farm	tulla coffee farm	diamond enterprise plc	snnp/kaffa zone,gimbowereda	50	60 kg	2014
10	Ethiopia	fahem coffee plantation		fahem coffee plantation	oromia	300	60 kg	2014
11	United States	el filo		coffee quality institute	antioquia	10	1 kg	2014
12	United States	los cedros		coffee quality institute	antioquia	10	1 kg	2014
13	United States (Hawaii)	arianna farms			kona	1	1	2009

Figure 4.1: Raw CSV data

Then the data needed to be normalized to fit the tables and their relations and constraints. The database design for the service responsible for serving the most amount of data, and to which the CSV data was mapped, looks as such: 4.2.

The data from the raw CSV files was translated into multiple files containing the corresponding SQL queries, with the help of a Python script: 4.3.

The API provides multiple endpoints, some targeted towards clients and some targeted towards an admin user. The admin user is mainly able to perform any sort of CRUD operations on all involved entities, such as adding, updating and/or deleting clients, countries, regions, beans etc. The clients are mainly able to query any coffee related data, such as countries, beans, inventories, orders, etc. They are also able to place orders, and when orders are placed or canceled the inventory gets updated accordingly. For ease of use and for presentation purposes, I have also developed a basic front-end application. This will provide a view and a way of interacting with the client endpoints.

For load testing, the focus is mostly on the client oriented endpoints, such as the ones for querying data or placing orders. Testing and comparisons on the admin specific endpoints are not performed, since in practice those are not going to be used that often, and will not have an impact on the overall performance of the application.

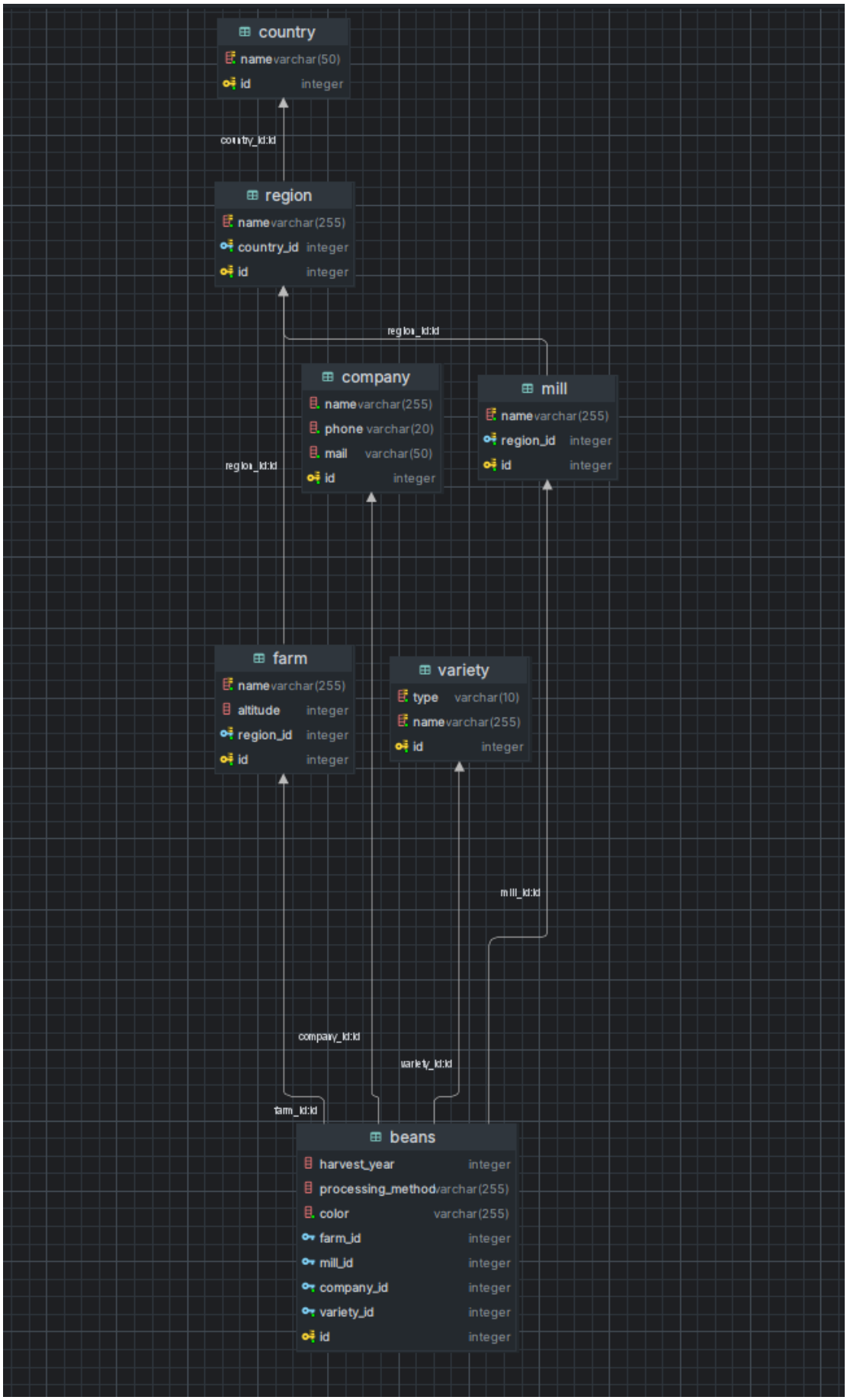

Figure 4.2: Partial database schema

```python
# VARIETY
varieties = df['Variety'].dropna().unique()
with open('insert_varieties.sql', 'w', encoding='utf-8') as f:
    for name in varieties:
        nr = random.randint(1, 10)
        type = 'ARABICA' if nr > 5 else 'ROBUSTA'
        f.write(f"INSERT INTO variety (name, type) VALUES ({sql_str(name)}, {sql_str(type)});\n")

# BEANS
with open('insert_beans.sql', 'w', encoding='utf-8') as f:
    for _, row in df.iterrows():
        harvest_year = row.get('Harvest.Year', None)
        harvest_year = int(harvest_year) if pd.notna(harvest_year) else 'NULL'

        processing_method = sql_str(row.get('Processing.Method', None))
        color = row.get('Color', None)
        color = sql_str(color if pd.notna(color) else 'Brown')

        farm = sql_str(row.get('Farm.Name', None))
        mill = sql_str(row.get('Mill', None))
        company = row.get('Company', None)
        company = sql_str(company if pd.notna(company) else 'Global Coffee Holdings LLC')
        variety = row.get('Variety', None)
        variety = sql_str(variety if pd.notna(variety) else 'Other')

        f.write(f"""INSERT INTO beans (
    harvest_year, processing_method, color, farm_id, mill_id, company_id, variety_id
) VALUES (
    {harvest_year}, {processing_method}, {color},
    (SELECT id FROM farm WHERE name = {farm} LIMIT 1),
    (SELECT id FROM mill WHERE name = {mill} LIMIT 1),
    (SELECT id FROM company WHERE name = {company}),
    (SELECT id FROM variety WHERE name = {variety})
) ON CONFLICT DO NOTHING;\n""")
```

Figure 4.3: Python script for translating CSV data into SQL queries

4.2 Architecture

Both implementations of the application were deployed and hosted in AWS, leveraging a lot of AWS' built-in solutions to create an efficient cloud-native architecture, especially when it comes to the microservices implementation. The monolith implementation is more or less functionally identical to the microservices one, with the exception that it encompasses all functionalities and resources in a single service, and it uses a single database.

Most resources belonging to the application are hosted in a VPC (virtual private cloud), in a specific AWS region, which corresponds to a geographical location such as Stockholm, Frankfurt, etc. This virtual private cloud acts as a local network, which can optionally be exposed to the internet.

On the public level, there is an application load balancer. All incoming HTTPS requests for both APIs first reach this load balancer on a public DNS name, which then routes them further based on their path or port. Incoming requests on the "/monolith" path are routed to the monolith implementation, which is technically just a single docker container with a lot of allocated CPU and memory resources, to mimic a single machine running the application. Incoming requests on the "/api" path correspond to the microservices implementation and are routed to a custom API Gateway, which takes care of further authentication, authorization, rate-limiting and routing. Incoming requests on port 8443 are routed to the OIDC server, either to its API through which clients can request tokens using their machine to machine credentials, or to its administration console which is accessible to an admin user for internal management.

The other services from the microservice implementation and their associated resources exist on the private level. If the request path is valid and the client can be authenticated, the API Gateway routes the requests to an internal application load balancer. This in turn routes requests based on their path to the corresponding running instances of each service.

All services, including the monolith, have an associated database. All databases are using Aurora Serverless engines, which permit the database to scale its capacity automatically in a configured interval, based on the incoming load. Each database can only be accessed internally by its corresponding service, and by an EC2 instance, which acts as a jump-host to allow remote database access using an SSH tunnel.

Communication between the running containers of the application and other AWS services is required in order for the containers to be able to create logs and metrics, to pull their corresponding docker image from private repositories, or to pull secrets hosted in AWS which are meant to be used as environment variables. This communication is kept private with the use of so-called VPC endpoints. These

ensure that communication between our VPC and global AWS services is kept private, and does not go through the public internet. Also, all communication inside the VPC or between the VPC and internet is managed by using an internal firewall, such that each resource is allowed to communicate only with the other resources it needs. For example, the API Gateway can only receive requests from the public load balancer, and send them to the internal load balancer. No other inbound or outbound traffic is allowed. Similarly, such firewall rules are configured for all the other resources. The overall architecture looks like this: 4.4.

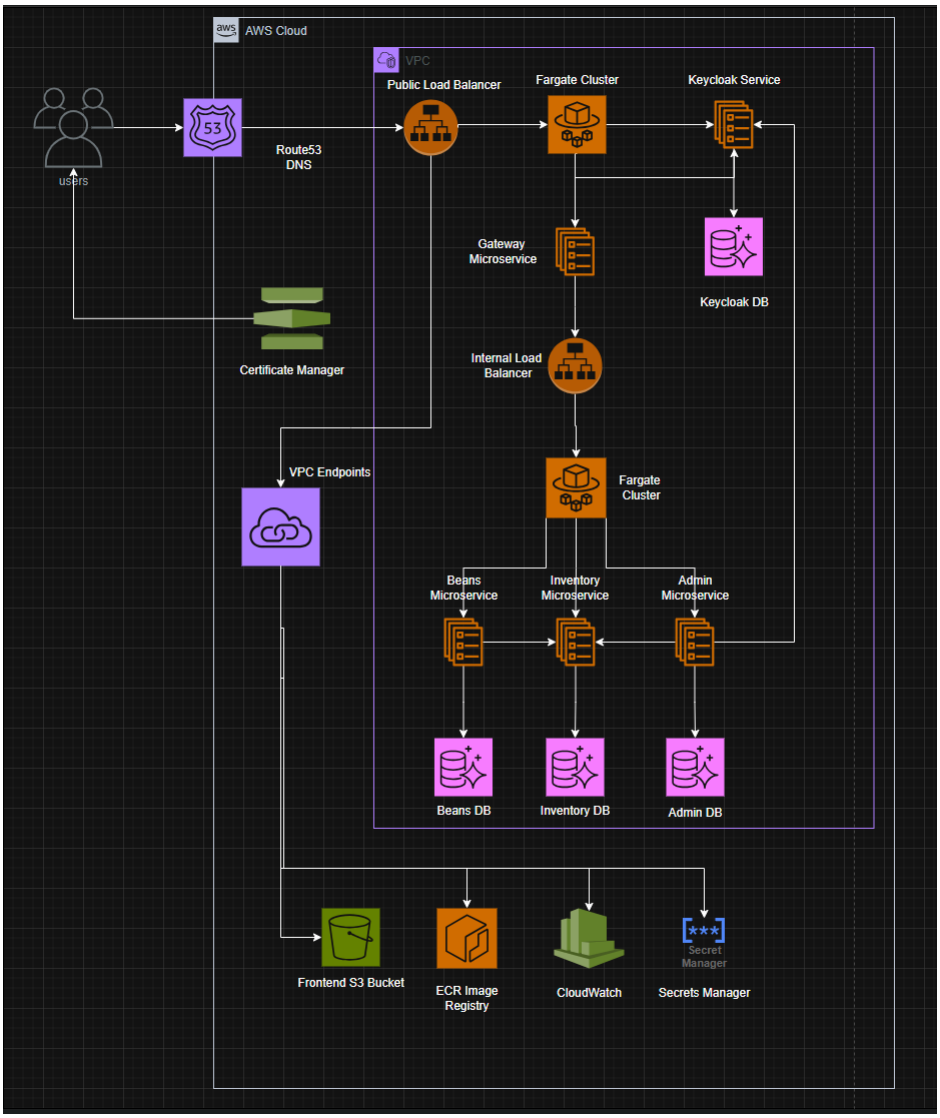

Figure 4.4: The AWS architecture of the microservices implementation

4.2.1 Backend

The backend of this application is composed of the previously mentioned APIs. The services which make up these APIs are split into two clusters, a so-called pub-

lic one and a private one. The public one contains the instances corresponding to the services which should be directly reachable from the internet. In contrast, the private one contains the instances which are only reachable internally.

Keycloak

- Purpose: This is basically the authentication and authorization server of the application. It is used to create clients, with associated scopes based on their subscription and to issue and validate tokens based on valid machine to machine credentials. Both its API and administration console are publicly reachable from the internet with HTTPS on the same URL as the rest of the application, but on a different port.
- Number of default instances: 1
- Allocated resources: 0.5 CPUs and 2 GB of memory
- Auto-scaling: None required. Since this service does not face much load, no auto-scaling is required.

Gateway

- Purpose: This service acts as an API Gateway for our application, and besides the authentication service, it is the only one reachable from the internet. All incoming requests to the API go first through this service, and are then routed to an internal load balancer, which takes care of further routing to the appropriate service. The authentication and authorization happen inside this service. Upon initialization it fetches the JSON Web Key Sets from the Keycloak service, and then uses them to validate the tokens and their associated scopes for each request. It also takes care of rate-limiting per client, using a bucket algorithm.
- Number of default instances: 1
- Allocated resources: 0.5 CPUs and 1 GB of memory
- Auto-scaling: between 1 and 2 instances. If the CPU usage across all instances exceeds 50% for 3 minutes, a new instance is created, and if it is below 40% for 15 minutes an instance is removed, if applicable. This helps ensure that there are always enough available threads to process all incoming requests. The percentage and the number of instances should suffice for even huge loads, since all requests are handled by non-blocking I/O threads.

Admin

- Purpose: This service is used by an admin user to create new subscription types, and to perform CRUD operations on API clients. All client

related operations performed here are also reflected in Keycloak, such as creating, enabling or disabling clients, or changing their scope based on their associated subscription. It also invokes the inventory service when deleting clients, to delete their associated orders.

- Number of default instances: 1
- Allocated resources: 1 CPU and 2 GB of memory
- Auto-scaling: None required. Since this service is only meant to be used by an admin user, it should not face any kind of load.

Beans

- Purpose: This service is the main client oriented service of the application. It manages data related to most resources of the application, such as countries, farms, coffee varieties, coffee beans, etc. It also invokes the inventory service to get the available stock and price when retrieving specific beans.
- Number of default instances: 1
- Allocated resources: 0.5 CPU and 1 GB of memory
- Auto-scaling: between 1 and 3 instances. If the CPU usage across all instances exceeds 60% for 3 minutes, a new instance is created, and if it is below 50% for 15 minutes an instance is removed. This approach ensures that the capacity can accommodate steady increasing loads over time. But in order to manage traffic spikes, which can happen instantly and be short-lived, if in any 1 minute interval the CPU usage for any instance exceeds 80%, a new instance is added.

Inventory

- Purpose: This service manages the inventory for all coffee beans and the orders placed by clients of the application. It is internally invoked by the other services to perform these operations.
- Number of default instances: 1
- Allocated resources: 0.5 CPU and 1 GB of memory
- Auto-scaling: between 1 and 3 instances. The auto-scaling rules are similar with the ones from the bean service, but since this service is invoked by the other services, we need to ensure high availability to avoid creating bottle-necks, so the thresholds are lower, 50% for the steady load auto-scaling, and 70% for the traffic spikes auto-scaling.

Monolith

- Purpose: This is functionally the equivalent of all the other services combined, managing all the data and doing its own rate-limiting and authentication and authorization, also with the help of the Keycloak service.
- Number of default instances: 1
- Allocated resources: 1 CPU and 2 GB of memory
- Auto-scaling: None required, since this is supposed to be a monolith application, which traditionally lacks auto-scaling.

For the build and deployment process, I've made use of the AWS ECR and ECS services, which provide support for storing docker images, and launching containers in predefined clusters. These containers can then, depending on the configuration, benefit from internal DNS name-spaces for service discovery, auto-scaling, monitoring, etc. After each microservice was implemented and tested locally, their corresponding artifacts were built using Gradle. Afterwards, for each service a docker image was created based on the above-mentioned artifacts, which was then pushed to a corresponding private docker repository, hosted in AWS ECR. These docker images, along with a task definition for each service, are required for the deployment process. The task definition is a JSON file, which specifies the usual configurations required for docker containers, such as the image that should be used, any required environment variables, logging drivers and configuration, port mappings, CPU and memory definitions, etc. These task definitions need to be uploaded to AWS ECS. Finally, using these task definitions, a configurable number of containers can be launched for each service.

4.2.2 Frontend

The frontend of this application is a simple client-side app developed in Angular. Its main purpose is to provide a mechanism to easily view and test the functionalities of the APIs, from a user perspective. Unlike the APIs, it was only deployed locally, not in the cloud. Based on the environment configuration provided during start-up, it can communicate with either the microservices API, or the monolith API, running either locally or in the cloud. This ensures that the same frontend can be used to test all the existing implementations of the APIs. The build and deployment processes are performed using the Angular CLI client, with commands such as "ng build" and "ng serve".

In terms of structure, the frontend was split into several components, one for each main functionality.

Home component This component functions as a container for the header of the page and the beans component.

Header component This component is the header of the page, always being present on the top side, after the user successfully logs in. It contains buttons for viewing the shopping cart, the user's orders and the beans page or for logging out.

Login component This component is the default one to which unauthenticated users are routed. It provides a login form. After successful authentication, users are redirected to the beans page.

Beans component This component provides the main view over most of the application's resources. It has drop-downs for countries, regions, farms, coffee variety types and companies, and a paginated table for the coffee beans. The drop-downs act as search criteria for the coffee beans, enabling the user to view only the desired beans, based on the provided values.

Bean details component This component represents a page containing details about a specific coffee bean. If in the previously mentioned table, the user selects a coffee bean, it will arrive on this details page. On this page, the bean can also be added to a shopping cart with the specified quantity.

Shopping cart component Here the user can see the items that were added to the shopping cart. The initially provided quantities for each item can be edited again here, and items can also be removed from the shopping cart. After validating the cart, the user can place an order on this page.

Orders component This component provides a view over the existing orders of the user. By default, only the in-progress orders are shown, but by changing the filtering criteria, also successful or canceled orders can be viewed.

4.3 Databases

When developing an application, usually the next natural step after defining the architecture is to design the database or databases needed for our services. A proper design has the potential to greatly improve the future performance of the app. But what is a proper design? It is a design that is specifically tailored for the needs and use cases of the application, and optimized for the most frequent operations we expect to happen, such as reads or writes.

The key database feature for ensuring performance are indexes. For basic applications, it is usually enough to rely on the default ones which are created such as primary keys or indexes corresponding to unique constraints if the logic of our tables demands them. But for more complex applications which may require high

amounts of write operations or very complex search queries spanning multiple tables, custom indexes are typically the go-to for improving performance.

Indexes should be placed only on fields which are used as search criteria in a lot of queries, or in complex queries which would otherwise be slower. Placing them on any other kind of fields does not really have much of an effect. Also, queries should be designed in a such a manner that the search is always performed on the indexed fields. This helps ensure that the defined indexes will actually be used when filtering or joining data.

The following example I encountered during the development of this thesis, showcases the difference between searching by indexed and non-indexed fields: 4.5.

```
postgres.public> select b.id, b.harvest_year, b.processing_method, b.color, f.name,
                        f.altitude, m.name, r.name, c.name, v.type, v.name, co.name from beans.beans b
                 inner join beans.variety v on b.variety_id = v.id
                 inner join beans.company co on b.company_id = co.id
                 left join beans.mill m on m.id = b.mill_id
                 left join beans.farm f on f.id = b.farm_id
                 left join beans.region r on m.region_id = r.id
                 left join beans.country c on c.id = r.country_id
                 where c.name = 'Mexico' and r.name = 'veracruz' and f.name = 'finca los barreales'
[2025-05-16 15:11:56] 4 rows retrieved starting from 1 in 158 ms (execution: 8 ms, fetching: 150 ms)
```

Figure 4.5: Query by indexed fields

As we can see in the example, the query uses indexes to traverse the whole table hierarchy, and only performs a table scan on the beans table, as it needs to select most of the fields from that table (4.6). An almost identical query yields higher total costs by about 20% and is visibly slower when replacing one of the search parameters with a non-indexed field, as showcased below (4.7). In this case, most of the indexes that were previously in-use are rendered obsolete, which creates a lot more table scans internally (4.8).

When dealing with tables and fields which are often updated, indexes should be considered with caution, since any write operation will also imply an index update, which can decrease the overall performance of the queries. After analyzing these aspects, I implemented all the queries in such a manner, that they always use an index, in order to ensure the future performance and speed of the app even under heavy traffic. The focus of these optimizations were mainly the read-oriented use cases, since those operations statistically happen way more than write operations, when browsing large lists of items and potentially placing orders.

Operation	Params	Rows	Actual Ro...	Total C...
Select				
Nested Loops (Nested Loop)		1	4	47.56
Nested Loops (Nested Loop)		1	4	39.38
Nested Loops (Nested Loop)		1	4	39.2
Nested Loops (Nested Loop)		1	4	39.0
Nested Loops (Nested Loop)		4	4	38.14
Hash Join		4	4	36.87
Full Scan (Seq Scan)	table: beans;	1291	1196	24.91
Transformation (Hash)		2	2	8.52
Bitmap Index Scan (Bi	table: farm;	2	2	8.52
Bitmap Index Scan	index: farm_name_regio...	2	2	4.29
Index Scan	table: mill; index: mill_pk...	1	1	0.32
Index Scan	table: region; index: reg...	1	1	0.21
Index Scan	table: variety; index: va...	1	1	0.2
Index Scan	table: company; index: ...	1	1	0.18
Index Scan	table: country; index: co...	1	1	8.17

Figure 4.6: Query analysis when using indexed fields

```
postgres.public> select b.id, b.harvest_year, b.processing_method, b.color, f.name,
                       f.altitude, m.name, r.name, c.name, v.type, v.name, co.name from beans.beans b
                 inner join beans.variety v on b.variety_id = v.id
                 inner join beans.company co on b.company_id = co.id
                 left join beans.mill m on m.id = b.mill_id
                 left join beans.farm f on f.id = b.farm_id
                 left join beans.region r on m.region_id = r.id
                 left join beans.country c on c.id = r.country_id
                 where c.name = 'Mexico' and r.name = 'veracruz' and f.altitude > 1000
[2025-05-16 15:14:59] 28 rows retrieved starting from 1 in 527 ms (execution: 13 ms, fetching: 514 ms)
```

Figure 4.7: Query by non-indexed fields

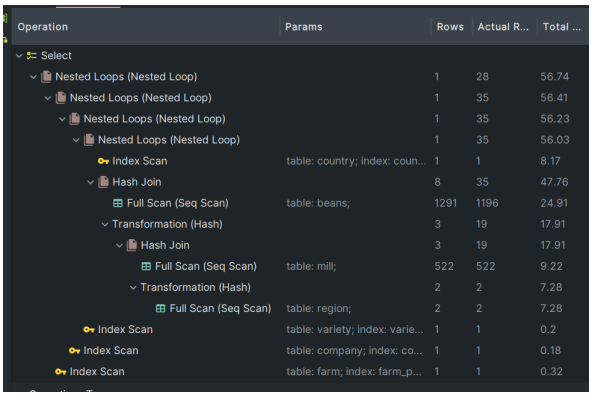

Figure 4.8: Query analysis when using non-indexed fields

4.3.1 ORM and query design at application level

Most modern applications nowadays use some form of ORM, or Object-relational Mapping. There are plenty of libraries and frameworks available for this, but for the scope of this thesis I will focus on Hibernate, which is one of the most popular and used ORM frameworks for Java.

ORM enables the programmer to seamlessly create mappings between classes in the application, also called entities, and corresponding database tables or views. Using Hibernate and other technologies from the Java ecosystem, such as Jakarta Persistence, one can define at a granular level how the entities should behave, what constraints and default values should be enforced, what fields should be included or excluded from insert and update queries, and how relations between entities, and implicitly tables, should be loaded, persisted or cascaded.

Because of this, a lot of the control over how queries are created and how and when they are performed is taken over from the programmer by the framework. This can be nice for development and testing purposes, but it can have its downfalls when the application and the persisted data grow.

For example, the most common way of handling and accessing relations is through annotations such as @OneToMany or @ManyToOne. These annotations define the direction of the relation and the owner. If a @ManyToOne annotation is used on a field in an entity, that means that in the underlying table there is a foreign key there.

The problem with @ManyToOne relations is that they are by default always eagerly loaded, which means that when fetching a set of entities, a subsequent select query will be performed for each eager relationship defined on those entities. One can imagine that depending on the size of the initial result set, this has the potential to create a lot of unwanted extra load, both on the database and on the application itself. Similarly, when using lazily loaded associations such as @OneToMany, each access to the associated collection generates an extra query. Depending on the use case and the number of associations, this also has the potential to generate a lot of unwanted queries and load. This is more commonly known as the ”n + 1” select issue.

There are a lot of options to mitigate these issues. One of the most recommended ones is to use an entity graph [Tho]. This is a mechanism that allows the developer to define a plan for a whole hierarchy of entities, with regards to how its relations should be fetched. This effectively enables the programmer to specify all the necessary joins that should happen in a query, so that the result set returned from the database already contains all the required data, avoiding the usual ”hidden” queries mentioned above. Thus, the total amount of queries is reduced to one. Multiple entity graphs can be defined and they can be optionally specified for each query,

allowing full control over all joins (4.9).

```
@NamedEntityGraphs({
        @NamedEntityGraph(
                name = Beans.ENTITY_GRAPH,
                attributeNodes = {
                        @NamedAttributeNode("variety"),
                        @NamedAttributeNode("company"),
                        @NamedAttributeNode(value = "farm", subgraph = "farm-subgraph"),
                        @NamedAttributeNode(value = "mill")
                },
                subgraphs = {
                        @NamedSubgraph(
                                name = "farm-subgraph",
                                attributeNodes = {
                                        @NamedAttributeNode(value = "region", subgraph = "region-subgraph")
                                }
                        ),
                        @NamedSubgraph(
                                name = "region-subgraph",
                                attributeNodes = {
                                        @NamedAttributeNode("country")
                                }
                        )
                }
        )
})
public class Beans {
```

Figure 4.9: Entity Graph Definition

Another issue that might accidentally happen with ORM technologies such as Hibernate, is when a query is translated from JPQL to SQL. All queries written with JPA or Jakarta Persistence are written in a sort of pseudo-sql, based on the application entities, which gets subsequently translated into SQL meant for the database. Each time an entity relation is accessed inside such a query, a join is generated in SQL. By accessing multiple relations sharing a common hierarchy, multiple unnecessary joins on the same hierarchy might be performed (4.10). Entity graphs also mitigate this issue, since all joins are already defined.

Taking this into consideration, I decided to run some benchmarks on these different approaches, in order to see which would be the most suitable for the application. The following table showcases the difference in the response time, for the same API request, for each of the mentioned scenarios. Even when running on a local environment, with small amounts of data and no load, the performance improvements are noticeable (4.1).

The results were pretty clear, which prompted me to use entity graphs for ensuring the minimum amount of queries and the proper fetching of entity hierarchies. Reducing the bottle-necks and potential performance issues at the database and ORM level was crucial for the development of this thesis. This was required in order to make sure that any differences in the load testing results between the

```
join
    beans.farm f1_0
        on f1_0.id=b1_0.farm_id
join
    beans.region r1_0
        on r1_0.id=f1_0.region_id
join
    beans.country c2_0
        on c2_0.id=r1_0.country_id
join
    beans.farm f2_0
        on f2_0.id=b1_0.farm_id
join
    beans.region r2_0
        on r2_0.id=f2_0.region_id
```

Figure 4.10: Duplicate Joins

Test scenario	Response time
Entity Graph	29 ms
N+1 Select Issue	33 ms
Lazy loading	35 ms
Duplicate Joins + Lazy Loading	38 ms

Table 4.1: Benchmarks

different API implementations are only influenced by the differences in architecture and the use of cloud technologies, and not the product of application issues or behaviour.

4.4 Use cases

The API has multiple endpoints for querying and managing data. The ones for querying data can be used by both clients and admin users. The ones for managing data can only be used by admin users. The base URL is the same for all endpoints as all requests made to this API are going through the previously mentioned API Gateway. All requests must have an authentication header present with a Bearer Json Web token. The query parameters are optional. If they are missing, all the items will be returned, otherwise the items will be filtered based only on the given parameters. The actions that can be performed by clients are the following:

Get countries

- Method: GET
- URL: /api/beans/countries

- Query parameters: name
- Result: A list of countries with name and id is returned

Get companies

- Method: GET
- URL: /api/beans/companies
- Query parameters: name
- Result: A list of companies with name, id, contact phone number and email is returned

Get varieties

- Method: GET
- URL: /api/beans/varieties
- Query parameters: name, type (can be ARABICA or ROBUSTA)
- Result: A list of varieties with name, id and type is returned

Get regions

- Method: GET
- URL: /api/beans/regions
- Query parameters: name, countryName
- Result: A list of regions with name and id is returned

Get farms

- Method: GET
- URL: /api/beans/farms
- Query parameters: name, regionName, countryName
- Result: A list of farms with name, id and altitude is returned

Get mills

- Method: GET
- URL: /api/beans/mills
- Query parameters: name, regionName, countryName
- Result: A list of mills with name and id is returned

Get beans

- Method: GET

- URL: /api/beans
- Query parameters: countryName, regionName, varietyName, varietyType, companyName, pageNumber, pageSize
- Result: A paginated list of beans with the following properties: id, harvestYear, processingMethod, color, farm, altitude, mill, region, country, company, varietyType, varietyName
- Notes: The pageNumber and pageSize query parameters are mandatory for pagination. Since this endpoint has the potential to serve a lot of data, it can only do so with pagination

Get bean details

- Method: GET
- URL: /api/beans/{beanId}
- Result: the same properties returned for the paginated list of beans on the get all endpoint, containing also the available quantity and the price per unit

Place order

- Method: POST
- URL: /api/inventory/orders
- Request payload: a list of order items composed of the ids of the beans the client wants to order, and the corresponding quantity
- Result: A new order will be created. Inventory quantity for the ordered beans will be updated based on the given quantity. If not enough beans are in stock an error response will be returned to the client with an appropriate message and the order will not be placed

Get orders

- Method: GET
- URL: /api/inventory/orders
- Result: A list with all of the client's orders will be returned. Each order will be composed of the following properties: the order status, the total cost of the order, the order number and a list of the ordered items, each item containing the bean id and the ordered quantity

Besides the above mentioned client endpoints used for querying data, there are also admin-only endpoints for inserting, updating and deleting data corresponding to all the mentioned resources, and also for managing clients. They all have a similar format. A couple of examples:

Create client

- Method: POST
- URL: /api/admin/clients
- Request payload: name, e-mail and subscription type, which can be either BASIC or PREMIUM
- Result: A client is created in the internal database, and also in the authentication provider, with a scope matching its subscription type. The client credentials for machine to machine authentication are returned to the user.

Delete client

- Method: DELETE
- URL: /api/admin/clients/{id}
- Result: The client with the associated id is removed from the internal database and also from the authentication provider. No further tokens will be able to be issued for this client. Also, any associated fulfilled or canceled orders belonging to the client will be deleted. If there are any in-progress orders associated with the client an error response will be returned to the user, so that the corresponding orders may be finalized first.

A typical client flow might from the API perspective might look like the following:

Create client The admin user first creates an account for a new client of the API. After the account is created, the credentials for using the API will be provided to the client: 4.11.

Get token Then the client needs to get a JWT using his credentials, in order to access and use the API: 4.12.

Get beans The API client might want to get all the available beans based on some search parameters. For example it might want to retrieve all the beans produced in the Veracruz region of Mexico: 4.13.

Get bean details Then the client might be interested in some of those beans and wish to see their details and inventory: 4.14.

Place order After checking the available quantity and the price for the beans, the client might place an order: 4.15.

View orders Finally, the client might check its in-progress orders: 4.16.

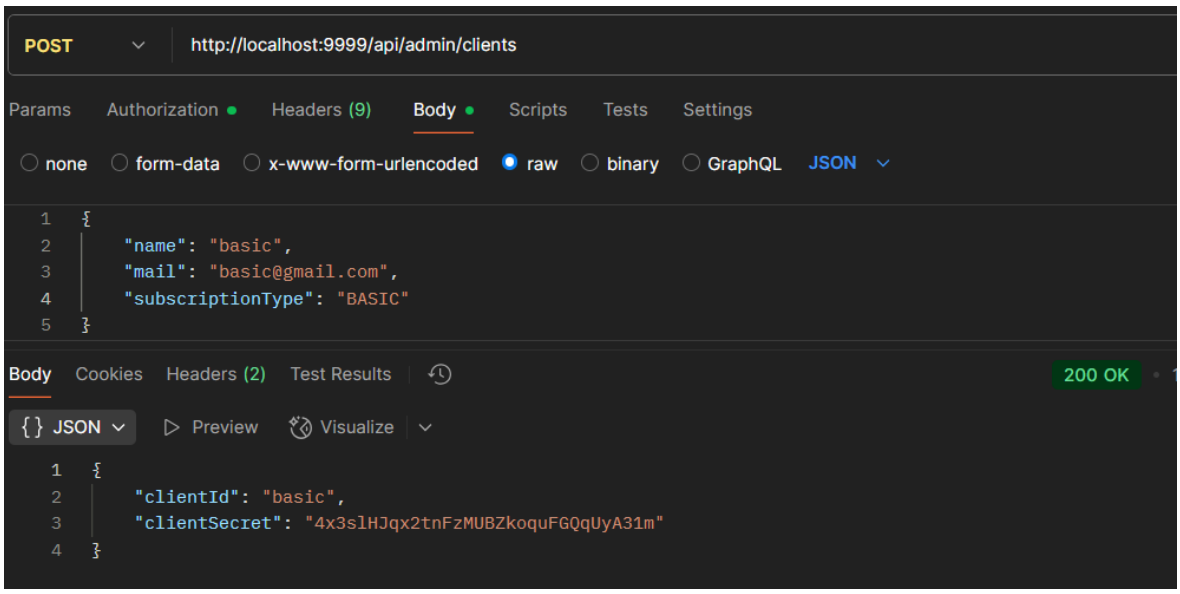

Figure 4.11: Admin user creates a client account for the API

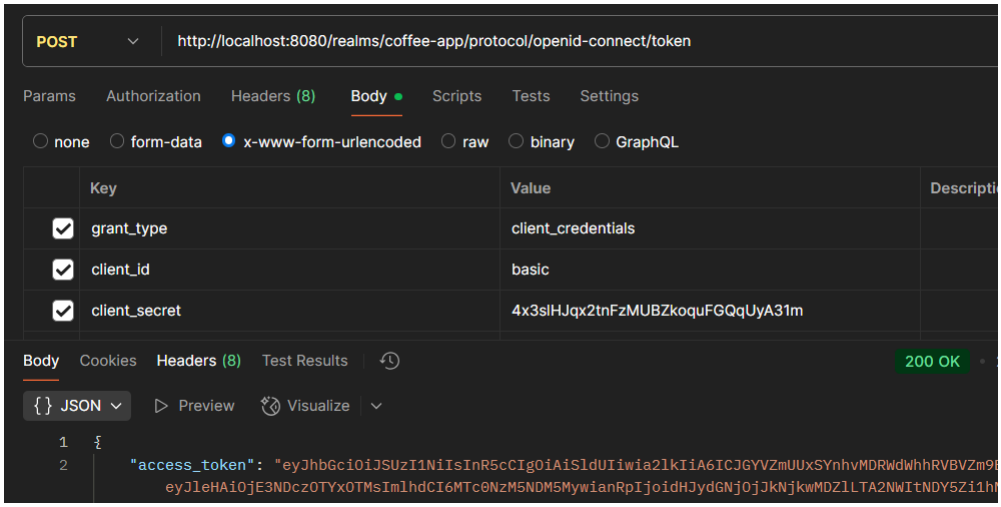

Figure 4.12: Client retrieves JWT using its machine to machine credentials

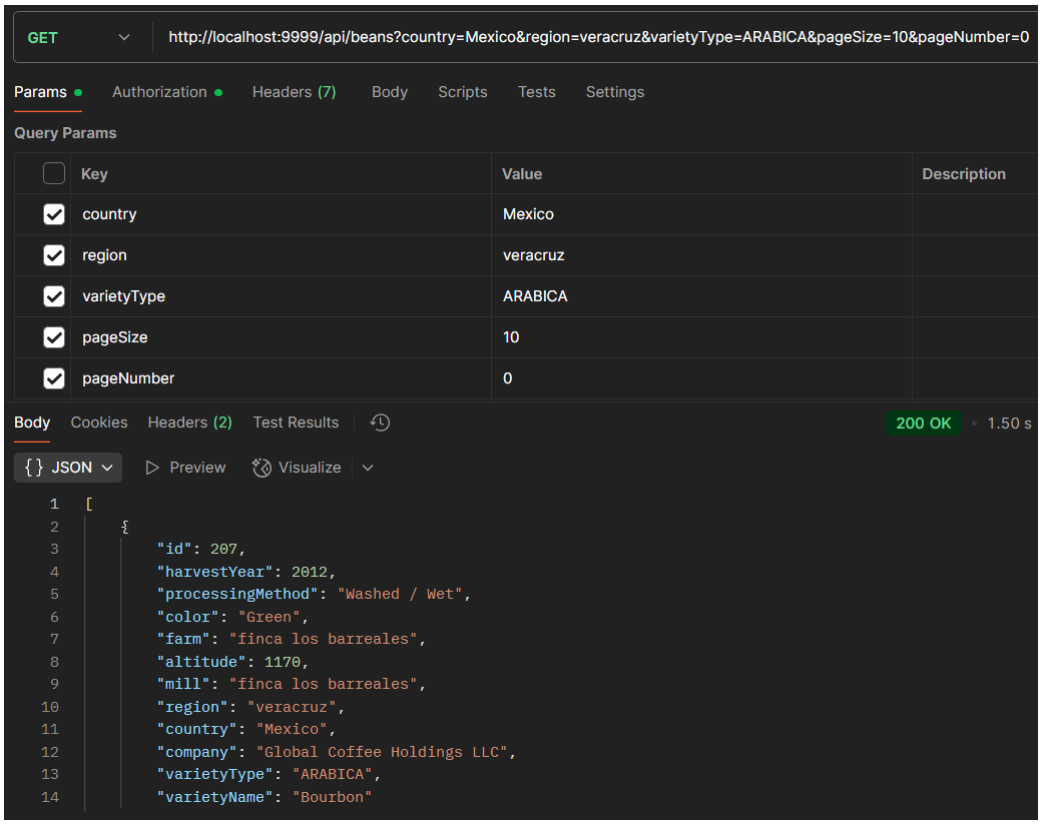

Figure 4.13: Client retrieves beans based on some search parameters

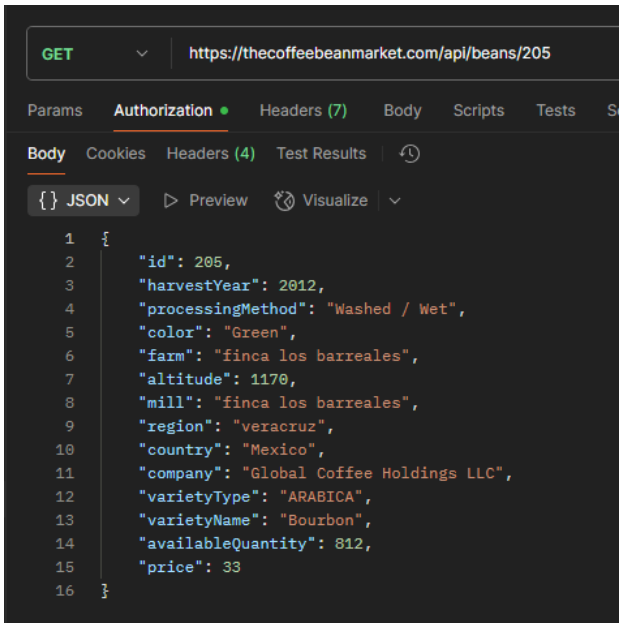

Figure 4.14: Client retrieves bean details

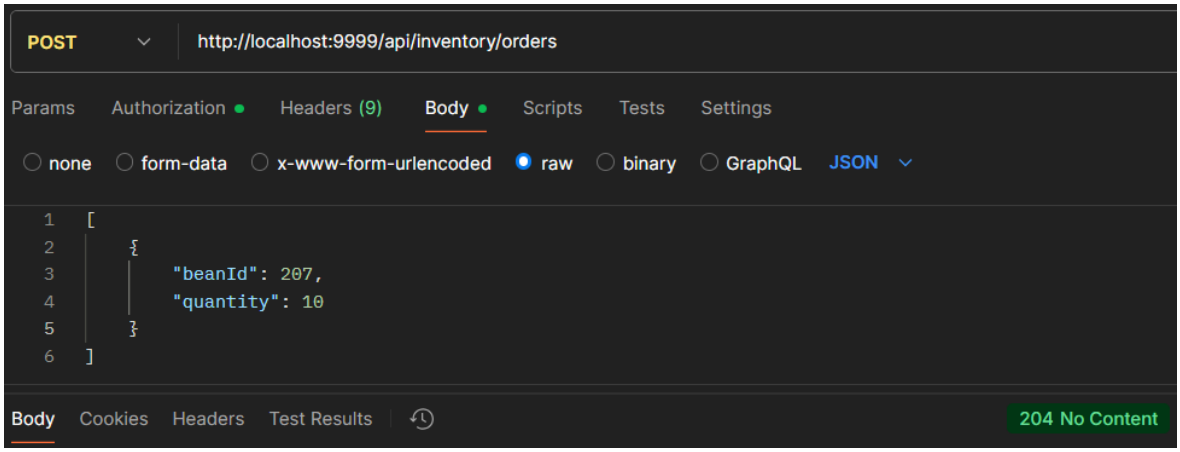

Figure 4.15: Client places an order for beans

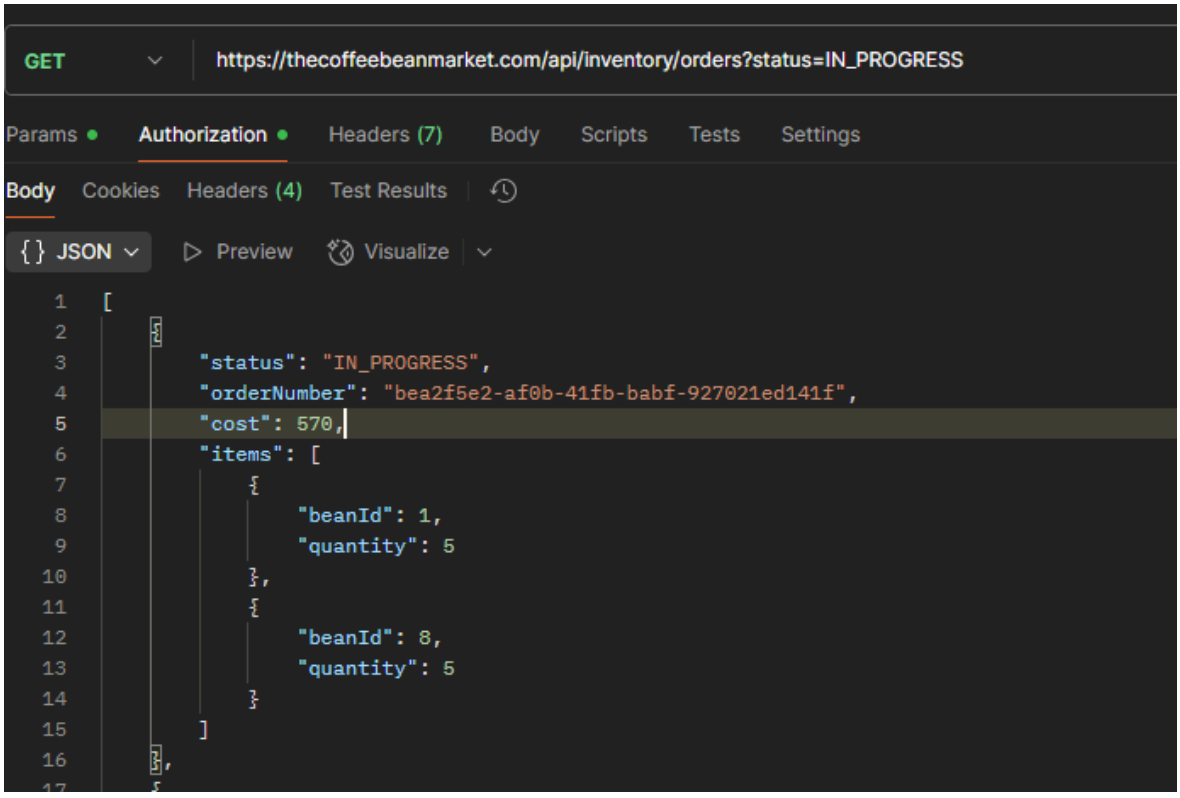

Figure 4.16: Client retrieves its in-progress orders

From the perspective of the frontend user, the flow and actions taken are quite similar to the steps listed in the description of the API use cases:

Log in Upon the first access to the app, the user will be prompted for its credentials in order to log in. It will need to provide a valid set of client credentials, which as previously mentioned, are issued when the admin user creates a new client for the API: 4.17.

View beans Then the user can view any beans in the application. By using the provided drop-downs for countries, regions and varieties, and paginated beans table, the user can easily and quickly search and view any beans that might be of interest: 4.18.

View bean details In the general beans table, the user can select any bean, which will redirect it to a bean details page. Here all information about the bean is displayed, including available quantity and price: 4.19.

Add bean to cart If the user wishes, it can add a specified quantity for the currently selected bean in the shopping cart: 4.20.

View cart and place order After adding some items to the shopping cart, these can be reviewed and updated by the user. Finally, after validating the contents of the cart, the user can place an order: 4.21.

View orders Then the user can check all of its existing orders. By default, only the in-progress ones are displayed, but also the successful and canceled ones can be viewed by changing the search criteria in the status drop-down: 4.22.

Log out After performing all desired actions, the user can log out, which will revoke its existing authentication token: 4.23.

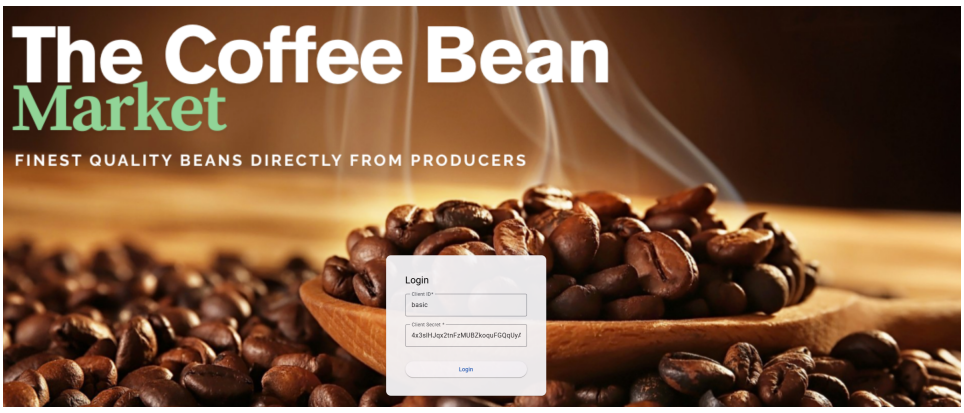

Figure 4.17: User logs in

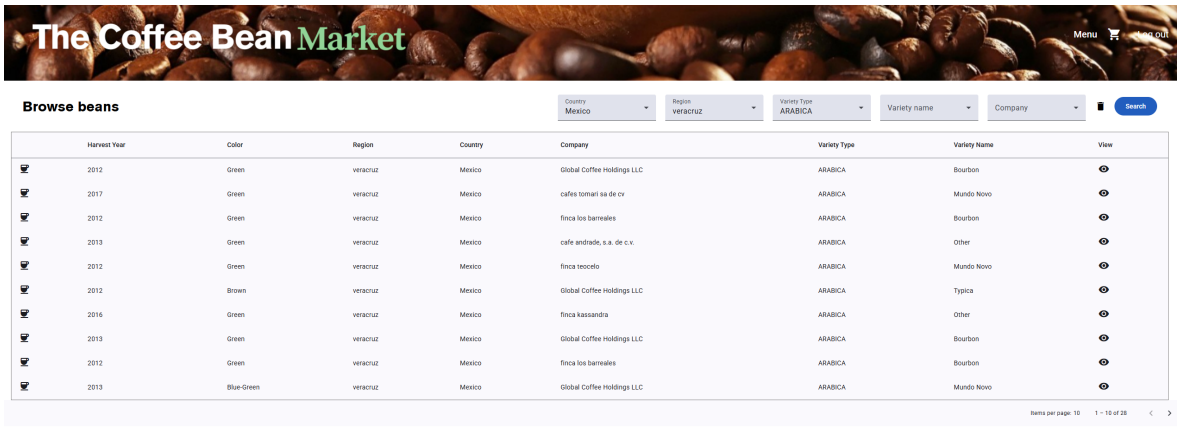

Figure 4.18: User searches for and views a list of beans

Figure 4.19: User selects a specific coffee bean and views its details page

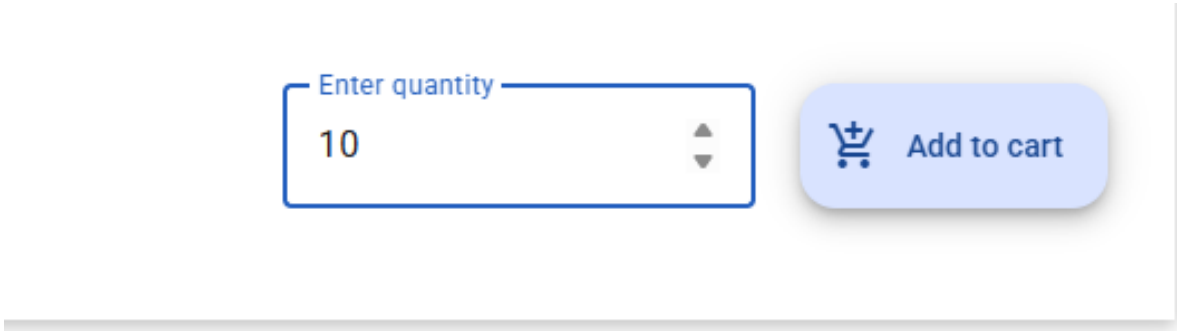

Figure 4.20: User adds a specific quantity for the selected bean to the shopping cart

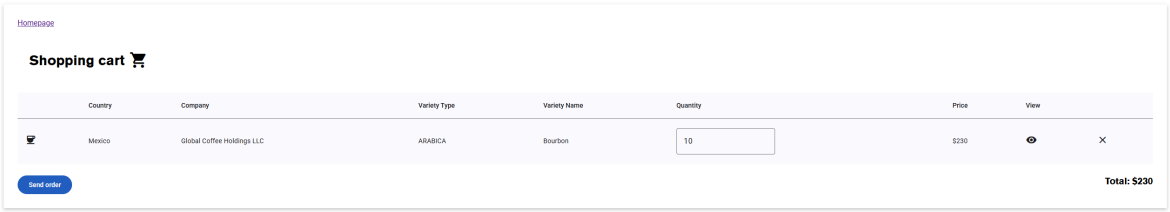

Figure 4.21: User views the shopping cart

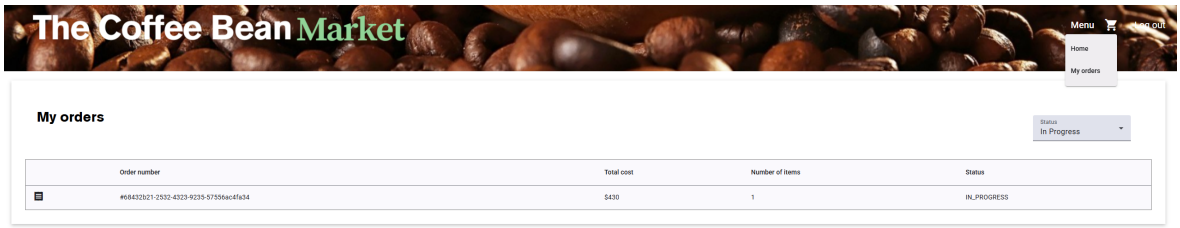

Figure 4.22: User views its in-progress orders

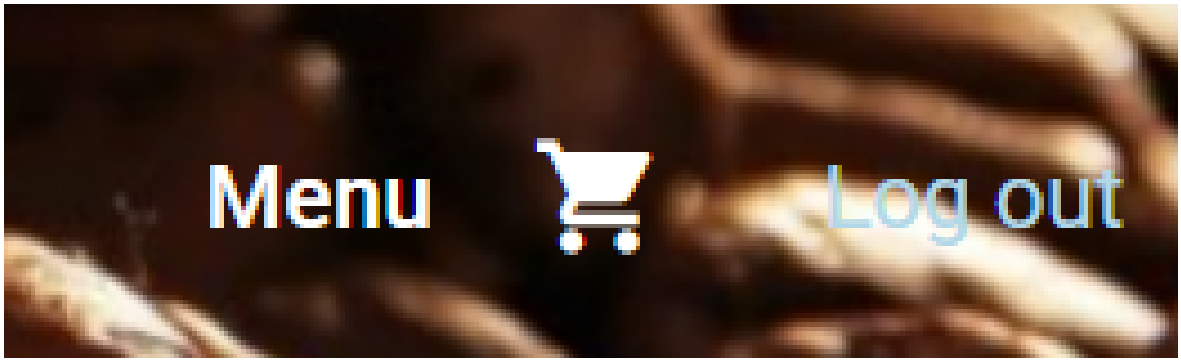

Figure 4.23: User logs out

4.5 Load testing and metrics

This section will highlight the performance differences between the two implementations, and what exactly led to these differences. This can be achieved with different load testing scenarios, with the purpose of seeing how the application behaves under specific traffic conditions, and what the breaking point of the application is.

To achieve this Artillery was used, which is a tool that effectively mimics user behavior based on some configuration and scripts. During testing, it creates multiple containers under a given AWS account, which then create multiple virtual users each second. Each of these virtual users ends up performing a given test scenario, which should represent a valid real life user flow. In the end, the test results are collected and reports are published to Artillery's cloud platform, which provides some nice dashboards and statistics for visualizing the results.

I have chosen two testing scenarios, one mainly focused on reading data, and one on writing data. The scenario for reading data should be the equivalent of a user basically just browsing the app and searching and viewing multiple coffee beans based on different search criteria. The writing scenario represents a user initially browsing the app for some time until the desired coffee beans are found and orders are placed.

Concretely, the actions performed by the virtual users during the tests for each of the scenarios are the following:

Read scenario

- Fetch token using client credentials
- Fetch all available countries and choose a random one from the returned list
- Fetch all available regions for the selected country, and choose a random one from the returned list
- Fetch coffee beans of either Arabica or Robusta varieties for the chosen country and region, in a paginated manner, with 10 coffee beans per page. This fetch is repeated until there are no more such coffee beans to be fetched
- For each retrieved coffee bean, fetch its details including available quantity and price
- Repeat this entire flow 5 times

Write scenario

- Fetch token using client credentials

- Fetch all available countries and choose a random one from the returned list
- Fetch all available regions for the selected country, and choose a random one from the returned list
- Fetch first 10 coffee beans of Arabica or Robusta variety for the chosen country and region
- If any beans were fetched, select a random one and place an order for it
- If no beans were fetched, repeat flow until an order is placed

Both of these scenarios are tested under the same virtual user load, which represents an initial traffic spike, which is then maintained for some time. For the first 30 seconds of the tests, the amount of users created per second grows from 1 to 10, and then for the following 120 seconds, 10 new users are created each second. This results in a load of 1255 active users using the app in parallel for about 3 minutes, which results in about 30 thousand requests for the read scenario, and 8 thousand requests for the write scenario.

The following table showcases the performance of the different API implementations for each scenario, with regards to the response time of the requests, the average request throughput, and the number of failed requests: 4.2.

Test scenario	Min	Avg	Max	Throughput	Failures
Microservices read scenario	3 ms	50 ms	601 ms	221 req/s	0
Monolith read scenario	1 ms	1.5 s	6.6 s	156 req/s	338
Microservices write scenario	4 ms	18 ms	717 ms	61 req/s	1
Monolith write scenario	4 ms	44 ms	2.6 s	59 req/s	1

Table 4.2: Performance benchmarking results

As expected, the microservices implementation outperformed the monolith one by quite a huge margin, especially in the read scenario. But if the implementations themselves are identical where do these differences originate from?

The main disadvantage of the monolith was that all the processing had to be done in one application, and all queries and data handled by one database. Even though it had double the CPU and memory resources of the microservices, the main bottleneck was that it had to perform double the amount of processing internally, and double the amount of queries on the same database, inside one transaction. This implicitly caused all transactions and all threads serving requests to be slower

as compared to its counterpart. Since the transactions were slower, database connections took longer until they were returned to the connection pool, which eventually led to the exhaustion of the connection pool, which led to many threads being blocked while waiting for available connections (4.24). This is also the reason why we can see 338 failed requests for the monolith implementation, because they resulted in timeouts. This wound up slowing down the whole application over time, since the request thread pool also shrank, minimizing the overall request throughput.

In comparison, each microservice performed smaller queries on their own database, resulting in faster transactions and requests, hence why the database connection pool and the request thread pool were never exhausted and the overall application could keep serving requests in an effective manner even under heavier load (4.25, 4.26). Another thing which worked in the favor of the microservices was the auto-scaling. Because of the small default quantity of allocated resources, initially the CPU usage (4.27) and request response time grew significantly. But after a bit of sustained load, auto-scaling kicked in, creating another instance of the beans service, which stabilized the CPU usage and decreased the response time (4.28).

So even though the microservices approach adds development and testing complexity and network communication overhead for each request, we can see that such an architecture, if implemented properly, can yield much better performance and resource consumption over time as the overall usage and load increase.

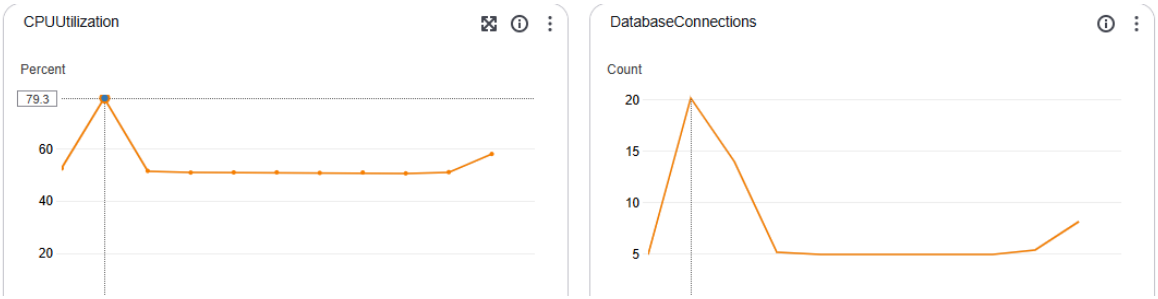

Figure 4.24: Exhaustion of database connection pool for monolith

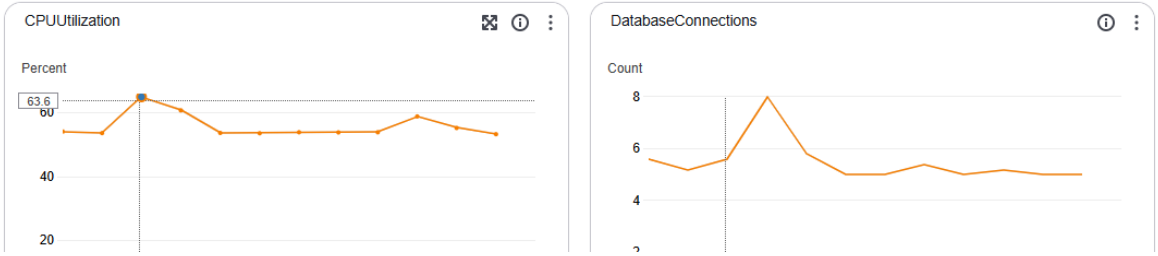

Figure 4.25: Database usage metrics for beans microservice

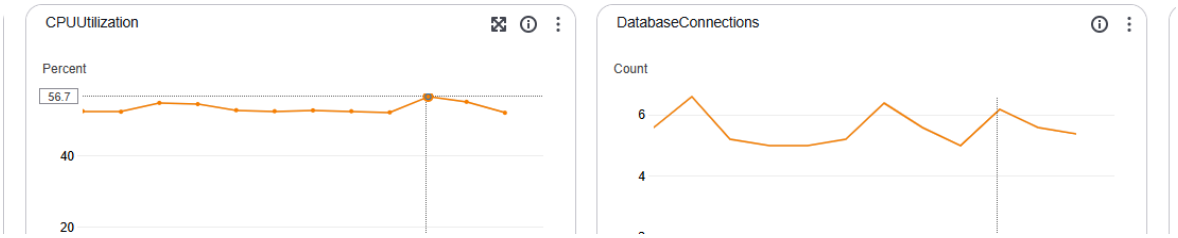

Figure 4.26: Database usage metrics for inventory microservice

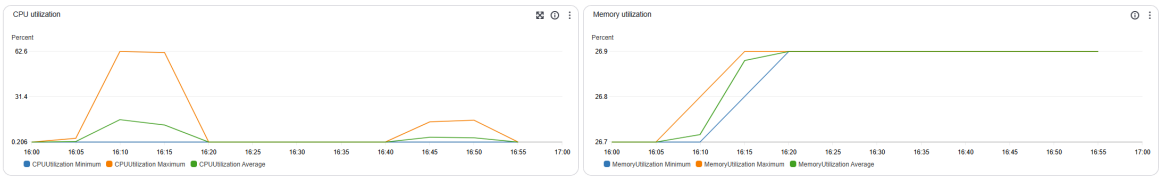

Figure 4.27: Usage metrics for beans microservice

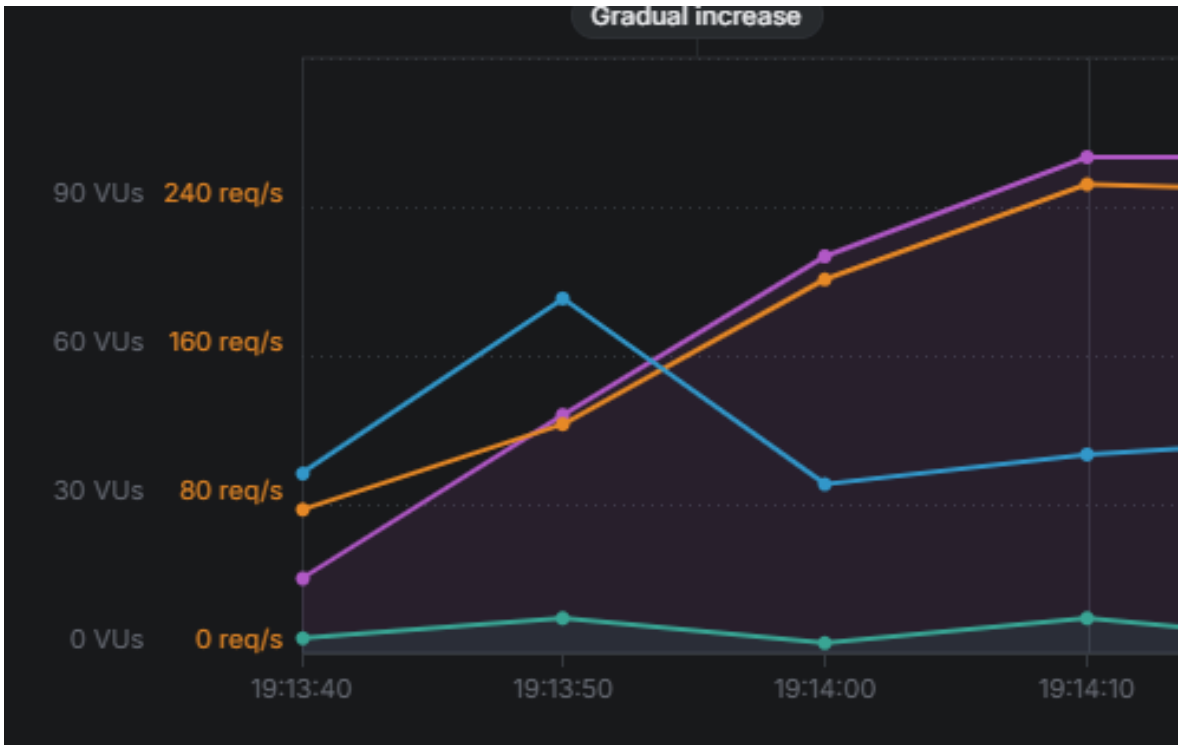

Figure 4.28: Evolution of request response time and throughput for microservice API

Chapter 5

Conclusions and future work

The aim of this paper was to provide a valid example of a performance optimization process for APIs meant to function on a global scale, simulating real-life scenarios. The performed comparisons showed significant differences between the microservices and the monolith, both with regards to response time and resource consumption, which are some of the most relevant metrics when measuring the performance and user satisfaction of any application. The performance improvements and benefits brought by cloud-native development and architecture are visible. The microservices API was able to sustain an initial traffic spike, followed by a significant steady load with an almost insignificant increase in response time and a minimal decrease in request throughput, certainly without affecting the user experience. In contrast, the monolith performed poorly and in a real-world scenario it would've definitely had a negative impact on the business, with increased costs because of a sudden spike in resource consumption, and with a poor user experience, since the response time grew and the request throughput shrank together with the applied load. The average response time exceeded 1 second, and there was even a small number of requests which resulted in timeouts. One can imagine how this behavior would worsen exponentially the more load was applied, and how it has the potential to drive users away from the application.

As future work and improvements, it would be worth to study the cost and performance differences between a serverless and a microservices architecture, since serverless is the new market trend when it comes to cloud-native architectures, and it could prove beneficial in certain contexts or use-cases.

Bibliography

[AK23] Marcus Rodriguez Srinikhita Kothapalli Jaya Chandra Srikanth Gummadi Arjun Kamisetty, Deekshith Narsina. Microservices vs. monoliths: Comparative analysis for scalable software architecture design. *Engineering International*, 2(2), 2023.

[CNC] Cloud Native Computing Foundation. `https://landscape.cncf.io/`. Online.

[CQI] Coffee Quality Institute. `https://database.coffeeinstitute.org/`. Online.

[Jam] James LeDoux. Coffee Quality Database. `https://github.com/jldbc/coffee-quality-database`. Online.

[OCO24] Adams Gbolahan Adeleke Lucy Anthony Akwawa Chidimma Francisca Azubuko Oyekunle Claudius Oyeniran, Adebunmi Okechukwu Adewusi. Microservices architecture in cloud-native applications: Design patterns and scalability. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*, 1(2), 2024.

[Pos] PostgreSQL: The World's Most Advanced Open Source Relational Database. `https://www.postgresql.org/`. Online.

[QUA] Quarkus Performance. `https://quarkus.io/performance/`. Online.

[Tho] Thorben Jannsen. JPA Entity Graphs: How to Define and Use a @NamedEntityGraph. `https://thorben-janssen.com/jpa-21-entity-graph-part-1-named-entity/`. Online.

[Tho20] Ravi Chandra Thota. Enhancing resilience in cloud-native architectures using well-architected principles. *International Journal of Innovative Research in Engineering Multidisciplinary Physical Sciences*, 8(6), 2020.

[Top] The top programming languages. `https://octoverse.github.com/2022/top-programming-languages`. Online.