Ballerina as a Language for Microservice Development and Deployment

2022 / 2023

1181846 Miguel Pereira

Ballerina as a Language for Microservice Development and Deployment

2022 / 2023

1181846 Miguel Pereira

Degree in Informatics Engineering

July 2023

ISEP Advisor: **Isabel Azevedo**

For Alexandra

Acknowledgements

To my wife Alexandra for the heroic effort in growing a proper family during the executing of an amount of work not conductive of such endeavours. To my children for the inspiration in motivation to continue working for you.

To my advisor, professor Isabel Azevedo, for the unwavering help and guidance given throughout this project without whom it would not have been possible to execute.

Resume

This project has the proposition of analysing the new Ballerina programming language as a possible new industry level candidate for the development of integrated microservice systems.

Its performance was analysed and reviewed through the design of a testing suite of various stress tests, made to figure out the performance during heavy usage of its services. Furthermore, comparison tests were made against the current common programming language C# of the .NET framework.

This showed that, performance-wise, the Ballerina language is a more than capable tool for the titular purposed of this work, and when compared to current industry solutions it does not show significant disadvantages in its use.

Keywords (Subject): Microservices, Cloud, Responsiveness, Systems testing, Stress Testing.

Keywords (Tech): Ballerina, Rest, Docker, Jmeter.

Index

Figure Index

Code Examples Index

List of Tables

Glossary

MVP	Minimum Viable Product
JS	Javascript
WS	Web service
POC	Proof-of-concept
API	Application Programming Interface
CRUD	Create, Read, Update, Delete – basic operations in API usage in regards of data manipulation
CPU	Central Processing Unit
CFD	Computational Fluid Dynamics
HTTP	Hyper Text Transfer Protocol

1 Introduction

1.1 Context

Ballerina is a modern programming language designed for building and deploying microservices, and it has recently joined the ranks of the top 100 most used programming languages. Its origins were in 2016 when WSO2, an enterprise software company, announced the project to supply a language specifically tailored to the challenges of distributed system development. Ballerina stands alongside other popular cloud-oriented languages widely adopted in the software development landscape.

For instance, Java is known for its portability and scalability, with a rich ecosystem of libraries and frameworks that eases cloud development. C#, developed by Microsoft, combines the power of the .NET framework with modern language features, making it well-suited for building cloud applications on the Microsoft Azure platform. Node.js, built on the JavaScript runtime, enables server-side JS execution, and has gained popularity for its event-driven, non-blocking I/O model, making it efficient for building scalable and real-time applications in the cloud. Ruby on Rails, often referred to as Rails, is a web application framework emphasizing developer productivity and convention over configuration. These languages, including Ballerina, exemplify the diverse set of tools available to developers for cloud-native development, catering to various requirements and preferences in the ever-evolving cloud landscape.

Since its introduction, Ballerina has gained recognition and popularity within the software development community. While it is still a relatively new language, its unique features and focuses on microservices have attracted developers seeking efficient solutions for building distributed systems. Ballerina’s adoption has been steadily growing, and it has recently achieved a significant milestone by entering the list of the top 100 most used programming languages.

The inclusion of Ballerina in the top 100 reflects its growing prominence and the recognition of its relevance in the rapidly evolving landscape of software development. Its presence among the most widely used languages signifies the increasing demand for specialized tools and languages tailored to specific domains, such as microservices and cloud-native architectures.

As software developers are interested in building cloud-based applications and working with microservices architecture, interest in Ballerina as a programming language, that could help tackle the challenges of building and deploying microservices-based applications, is growing. It is particularly interesting Ballerina's built-in support for key microservices-related features, which could help manage the complexity of these systems and focus on building the functionality that applications may require.

To explore Ballerina's capabilities and gain firsthand experience with the language, the projects proposal is to undertake the creation of an application for a sandwich shop with multiple services. This project allows working with Ballerina's microservices-oriented language constructs, such as annotations for defining service interfaces and endpoints, and support for asynchronous communication using events.

1.2 Proposed Issue

The goal of the project is to analyse Ballerina as a language for the development of microservices and their deployment, by developing an application that meets the needs of a sandwich shop, with multiple services, including sandwich creation, ingredient listing, and review management. The application will also include business-related features such as order placement and support for multiple languages.

To ensure the scalability and flexibility of the application, the architecture will be based on microservices. This means that each service will be designed to perform a specific function and will communicate with other services through APIs or events. The use of events will allow for asynchronous communication between services, which can help to reduce coupling and improve overall system performance.

Overall, the project's aim is to highlight the capabilities of Ballerina for building scalable and flexible microservices-based applications. By using a real-world example, the project can demonstrate how Ballerina can be used to build complex systems with multiple services and features. The open-source nature of the project will also make it easier for other developers to learn from the code and contribute to the Ballerina community.

1.2.1 Objectives

The practical aim of the project is to develop a microservices-based application using the Ballerina programming language to meet the needs of a sandwich shop. The application will consist of multiple services, such as Sandwich, Ingredient, and Review, among others. Each service will handle a specific set of features, such as creating sandwiches, listing available ingredients, placing an order, and providing multilingual support.

The application will be built using a microservices architecture, which is a modern approach to software design that emphasizes modularity, scalability, and resilience. In a microservices architecture, applications are broken down into small, independent services that can be developed and deployed separately. These services communicate with each other using lightweight protocols such as HTTP, and they can be scaled up or down as needed to meet changing demands.

The project will also make use of events for communication between services. Events are a powerful way to decouple services, making it easier to support and evolve the application over time. In a microservices architecture, events are typically used for asynchronous communication between services, allowing them to exchange information without being tightly coupled.

The project will also use the Ballerina language, as a constraint for analysis, and one other language, C#.NET, for comparison whilst under load, and must use most of the features and language specifics utilities that make the core of the tools available.

Finally, the project will make all its work available in a public repository. This will contribute to the dissemination of Ballerina programming language, its characteristics, and potential difficulties and problems that may condition its adoption. By sharing its work with the broader development community, the project aims to help others learn about Ballerina, its strengths, and weaknesses, and how it can be used to build microservices-based applications.

1.2.2 Approach

To start a project in a new language, much like any other tool, a period of familiarization with the assets at hand was needed. To this end several tests in a POC, proof-of-concept, style was realized, where it was tested the techniques to create http services, interaction techniques, both in REST and gRPC, and Java interop calls to existing libraries.

From this was gathered all the necessary knowledge to create workable services that are a good representation of the Ballerina language, its usability as a development tool, and the mechanisms for deployment in a micro service fashion.

Next tests are devised in three separate ways. Unit testing, using the framework of the language itself, integration tests using Postman as the tool to carry out this, and load/stress testing using Jmeter to respect the non-functional requirements of the project.

For comparison's sake, one of the services on the system is replicated in another tech stack, in particular .NET C#, to have a baseline against a modern and common API and microservices development technology.

1.2.3 Contributions

By sharing my work publicly in a public repository, I hope to contribute to the Ballerina community and help others learn about the language's features, potential difficulties, and benefits. Ultimately, this project is as an opportunity to further skills as a developer and gain practical experience working with a language that could be useful in future projects.

In addition to building the application, the project will also contribute to the dissemination of the Ballerina language by making all work available in a public repository. This will include documentation, source code, and other resources that can help other developers learn about the language and its potential uses.

1.2.4 Work planning

The project will be structured around a series of developer milestones, each of which will focus on a specific set of goals. By breaking the project down into smaller, more manageable pieces, we can ensure that we are making steady progress towards our goal of delivering a fully functional microservices-based application.

The first milestone will focus on developing the basic infrastructure of the application, including the microservices architecture and the communication protocols between services. This will involve setting up the necessary tools and frameworks, such as Ballerina and Docker, and designing the high-level architecture of the application. At the end of this milestone, we should have a basic skeleton of the application up and running, and we should be able to communicate between the services.

The second milestone will focus on developing the core functionality of the application, including the ability to create sandwiches, list ingredients, and place orders. It will also start the implementation of some basic multilingual support, to ensure that the application can be used in a wide range of languages. At the end of this milestone, we should have a fully functional MVP of the application, which can be used to create and order sandwiches.

The third milestone will focus on refining and expanding the functionality of the application. This may involve adding new services, such as a Review service, or integrating with other languages or technologies if possible. We will also perform load testing of the full system to ensure that it can handle the expected amount of traffic, and responsive testing of the individual APIs to ensure that they are performing correctly.

For each milestone, we will perform a full set of testing, including load testing of the full system and responsive testing of the individual APIs. This will help to find and address any performance or scalability issues early in the development process before they become more difficult and expensive to fix.

1.3 Report structure

This report, written for explaining the context and objectives of this project, has 5 chapters. This first one is an introductory presentation to the work, its purposes and objectives. The second holds the explanation of the main tool for the execution of work, the Ballerina language and the following third contains the analysis of the problem and first approach on how to solve it. The last two chapters, the 4th and 5th explain the implementation methods and conclusions derived from them, respectively.

2 State of the Art

2.1 Microservices

2.1.1 API Gateway Architecture

An API Gateway is a key component in modern software architectures, particularly in the context of microservices and distributed systems. It acts as a centralized entry point for all client requests to a set of backend services or APIs.[2] Its primary purpose is to supply a unified interface and manage the interactions between clients and the underlying services.

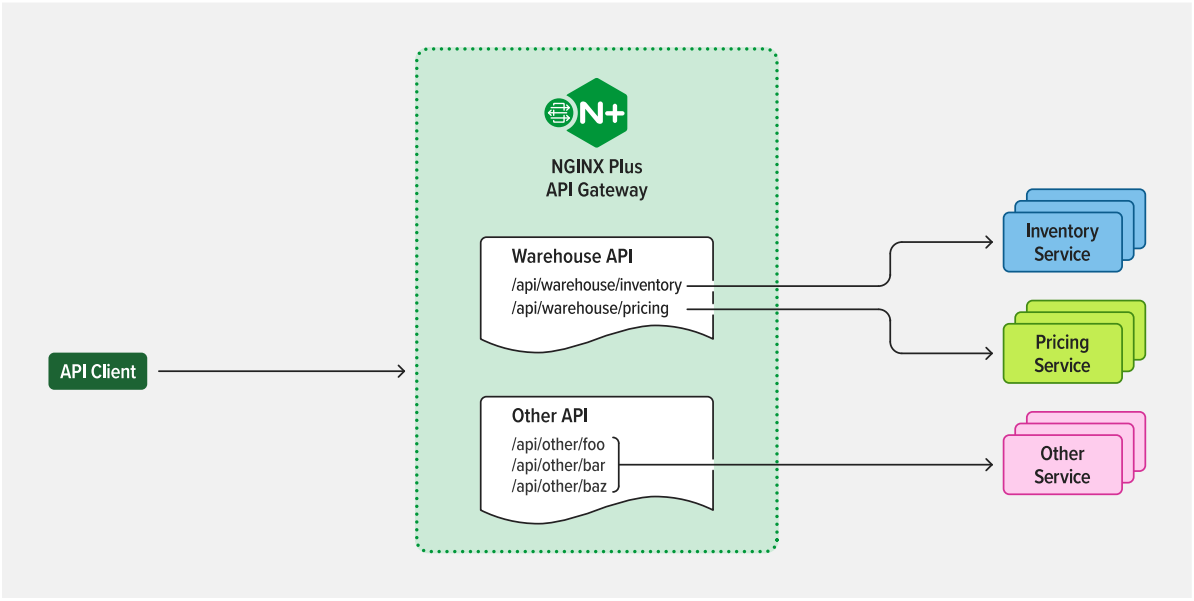


Figure 1 - Api Gateway example retrieved from NGINX [12]

The main uses of an API Gateway include:

1. **API Aggregation:** An API Gateway can aggregate multiple backend APIs into a single, cohesive API. This allows clients to interact with a unified API surface, even if the underlying services have different protocols, formats, or versions. The API Gateway handles the complexity of routing requests to the right services and consolidating responses.
2. **Protocol Translation:** Different clients may use different protocols to communicate with backend services. The API Gateway can handle protocol translation by

accepting requests in one protocol (e.g., HTTP/REST) and translating them into the protocol understood by the backend service (e.g., gRPC or SOAP).

3. **Security and Authentication:** API Gateways often handle security-related concerns, such as authentication and authorization. They can enforce access control policies, confirm API keys or tokens, and ensure that clients have the necessary permissions to access specific resources.
4. **Rate Limiting and Throttling:** API Gateways can implement rate limiting and throttling mechanisms to protect backend services from excessive traffic. By enforcing limits on the number of requests or the rate of requests, the API Gateway helps prevent service overloading and improves overall system stability and performance.
5. **Caching:** To enhance performance and reduce backend service load, an API Gateway can cache responses from the underlying services. It can store commonly accessed data and serve later requests directly from the cache, eliminating the need to forward requests to the services unless necessary.
6. **Monitoring and Analytics:** API Gateways supply a central point for monitoring and gathering analytics about API usage. They can collect metrics, logs, and traces, which can be analysed to gain insights into system performance, find bottlenecks, and track usage patterns.

Overall, an API Gateway simplifies the management of complex distributed systems by supplying a single-entry point, handling protocol translations, enforcing security, and offering additional features like rate limiting and caching. It improves system scalability, resilience, and security while providing a streamlined experience for clients interacting with the underlying services. Next is a more detailed exposition about the two main techniques used in project.

2.1.1.1 Aggregator Pattern

The aggregator pattern is a common technique employed within an API gateway to ease the composition of different APIs and the aggregation of data from multiple services. This pattern allows the API gateway to act as a mediator between clients and the underlying microservices, enabling the consolidation of data and functionality. By using this design choice, the API gateway can receive

requests from clients and orchestrate interactions with multiple microservices to fulfil those requests. The gateway can make parallel or sequential calls to various microservices, collecting the necessary data from each service. It then combines or aggregates the data into a cohesive response that meets the specific requirements of the client.

The aggregator pattern supplies several benefits. Firstly, it reduces the number of requests made by the client, minimizing network overhead, and improving overall performance. Instead of the client making multiple requests to different microservices, it can make a single request to the API gateway, which handles the data retrieval and aggregation on its behalf.

Furthermore, the aggregator pattern enables the API gateway to transform or reshape the responses from the microservices to align with the specific requirements of the client. It can perform data transformation, filtering, or formatting operations, ensuring that the client receives the desired response structure and content.

It's worth noting that while the aggregator pattern offers advantages in terms of reduced network traffic, simplified client interaction, and response customization, it also introduces considerations and potential challenges. The API gateway must handle error handling, fault tolerance, caching, and synchronization of data from multiple sources. Additionally, the performance of the aggregator can become a bottleneck if not properly optimized, as it needs to efficiently manage multiple concurrent requests and handle the aggregation process.

In summary, the aggregator pattern is a powerful technique employed within an API gateway that allows for the composition of different APIs and the aggregation of data from multiple services. By leveraging this pattern, the gateway can simplify client interactions, reduce network overhead, and customize responses to meet specific client requirements.

2.1.1.2 Dedicated per Service Database

By adopting a single database for each microservice, data management becomes more manageable and efficient. Each microservice can have its own dedicated database, which enhances data isolation and ensures that the services can evolve independently without affecting other services. This approach aligns well with the microservice principles of autonomy and decoupling. Additionally, separate databases help scalability and performance optimization for individual services, as each database can be scaled and optimized based on the specific needs of its corresponding microservice.

However, it's important to note that adopting a single database for each microservice also introduces challenges, such as data consistency across services and potential duplication of data. Proper data

synchronization mechanisms and data sharing strategies need to be implemented to ensure the consistency and coherence of the overall system.

2.1.2 API Testing

API Load Testing, Soak Testing, and Spike Testing are essential techniques used in performance testing to evaluate the behaviour and stability of an API under different scenarios. Each test type focuses on specific aspects of an API's performance characteristics.

API Load Testing involves subjecting an API to varying levels of simulated user traffic to assess its performance under expected or peak loads. The goal is to figure out the API's capacity and response times while maintaining acceptable performance levels. By generating a high volume of concurrent requests, load testing helps identify performance bottlenecks, such as high response times or resource limitations. It supplies insights into the API's ability to handle the anticipated user load and helps optimize its performance.

Soak Testing, also known as endurance testing or longevity testing, evaluates the API's stability and reliability over an extended period. The API is subjected to a sustained workload to check its behaviour over time and identify any degradation in performance or resource leaks. Soak testing is particularly useful for uncovering issues like memory leaks, database connection leaks, or other gradual performance degradations that may not be clear during shorter duration tests. It helps ensure the API's stability and robustness under continuous usage.

Spike Testing involves subjecting the API to sudden and extreme increases in user traffic to assess its ability to handle rapid load fluctuations. This test simulates scenarios where the API experiences sudden spikes in user activity, such as during marketing campaigns or sudden bursts of traffic. The aim is to evaluate how the API handles the surge in requests and whether it can gracefully recover to maintain optimal performance. Spike testing helps show any scalability limitations, performance bottlenecks, or vulnerabilities that may arise under such high-stress situations.

In summary, API Load Testing assesses an API's performance under varying user loads, Soak Testing evaluates its stability and reliability over an extended duration, and Spike Testing tests its ability to handle sudden load spikes. These testing methodologies collectively help find performance issues, ensure scalability, and optimize the overall performance of an API. By conducting these tests, organizations can gain confidence in their API's performance and deliver a high-quality experience to users under different usage scenarios.

2.2 Ballerina

2.2.1 Overview

Ballerina is a modern programming language that was specifically designed for building cloud-native applications. It was first released by WSO2, an enterprise open-source integration company, and it has since gained popularity among developers who are looking for a simpler, more productive way to build microservices and other distributed systems.[5]

```
resource isolated function post name/list(string[] ingredient_names) returns Ingredient[]|error {
    Ingredient[] list = [];
    foreach string name in ingredient_names {
        list.push(check getIngredientByName(name));
    }
    return list;
}
```

Code Example 1 - Resource function that lists ingredients [1]

2.2.2 Language Features

2.2.2.1 Basic Features

As a programming language designed for building cloud-native applications with a strong emphasis on integration and distributed systems. It incorporates several features that make it well-suited for these domains. These are some of the basic features of Ballerina:[6][8]

1. **Concurrency Model**: Ballerina supports a concurrent programming model, allowing developers to write scalable and efficient code. It supplies built-in support for handling concurrent execution using lightweight parallelism and message passing.
2. **Strong Typing**: Ballerina is a statically typed language that enforces strong typing at compile-time. It includes a rich type of system with support for primitive types, structured types (such as objects, records, and tuples), and union types. Type safety helps catch errors early and promotes robust code. As seen in the example above, the return type is a Union of a list of domain objects and the base error type.
3. **Service-Oriented Architecture**: Ballerina is designed to build services as a core part. It supplies a native syntax for defining services, including service endpoints, operations, and resource functions, as seen in the example above. Services can be easily exposed via different protocols such as HTTP, gRPC, and WebSocket.
4. **Built-in Concurrency Constructs**: Ballerina offers constructs like workers and isolated functions to ease concurrent programming. Workers enable concurrent execution of tasks,

while isolated functions ensure that shared data is protected from concurrent modifications, promoting safe concurrent programming.

5. **Integration and Connector Libraries**: Ballerina comes with a rich set of built-in connectors and libraries for seamless integration with various systems and technologies. It provides out-of-the-box support for connecting with databases, message brokers, RESTful services, cloud platforms, and more.
6. **Message-Driven Architecture**: Ballerina adopts a message-driven architecture, where interactions between services are modelled as message exchanges. It provides constructs like listeners, responders, and connectors to help message routing, transformation, and mediation.
7. **Security**: Ballerina offers built-in security features to ensure secure communication and data handling. It supports secure socket communication, encryption, digital signatures, and authentication mechanisms, allowing developers to build secure and robust applications.
8. **Tooling Support**: Ballerina comes with a comprehensive set of developer tools to enhance productivity. It includes an integrated development environment (IDE) with code completion, debugging, and other productivity features. It also offers testing frameworks and packaging utilities to support the software development lifecycle.

2.2.2.2 Error Handling

Error handling is part of the base language features of Ballerina, where a robust error handling mechanism is provided to effectively manage and propagate errors. This mechanism revolves around the use of the 'error' type and the 'check' keyword, enabling developers to handle and communicate errors in a structured and controlled manner.

The ***'error'*** type in Ballerina is a fundamental construct specifically designed to stand for error conditions. It serves as a distinct type that allows developers to explicitly define and handle errors within their code. When meeting an error condition, Ballerina programmers can create an error using the ***'error'*** constructor function and provide an error message as an argument. For instance, the statement "*error customError = error("This is a custom error");*" creates an error object named *'customError'* with the message "This is a custom error". This explicit representation of errors ensures that they can be easily found and managed within the codebase.

To handle errors and propagate them in a controlled manner, Ballerina introduces the '***check'*** keyword. It is typically used within functions or code blocks that may generate errors and need to communicate them back to the caller. By employing the ***'check'*** keyword, developers can explicitly check for potential errors and return them as part of the function's return type. [9]

As seen in Code Example 1, the ***'check'*** is used to confirm that there is no error with any of the elements that compose the list to be returned in that request.

2.2.2.3 Concurrency

In Ballerina, concurrency is supported using workers, which are lightweight execution units that enable concurrent execution of tasks. Workers provide a structured and efficient way to handle parallelism and asynchronous operations.

```
import ballerina/io;

public function main() {
    worker A {
        int num = 10;

        // Sends the `10` integer value to the `B` worker asynchronously.
        num -> B;

        // Receives the `Hello` string from the `B` worker.
        string msg = <- B;
        io:println(string `Received string "${msg}" from worker B`);
    }

    worker B {
        int num;

        // Receives the `10` integer value from the `A` worker.
        num = <- A;
        io:println(string `Received integer "${num}" from worker A`);

        // Sends the `Hello` string to the `A` worker asynchronously.
        string msg = "Hello";
        msg -> A;
    }
}
```

Code Example 2 - Full worker use example [76]

The following is an explanation of how workers work in Ballerina:

Worker Creation: Workers are created using the worker keyword followed by an identifier.

```
worker A { … };
worker B { … };
```

Worker Invocation: Workers are invoked using the start statement, followed by the worker identifier and the name of the function or expression to execute in the worker.

```
num -> B;
```

Concurrency and Communication: Workers execute concurrently, enabling parallel execution of tasks. They communicate with each other and the main execution flow

```
// Receives the `10` integer value from the `A`
worker.
num = <- A;
io:println(string `Received integer "${num}" from
worker A`);
```

using message passing. Ballerina supplies the *'worker.send'* and *'worker.receive'* constructs for inter-worker communication. The *'worker.send'* statement is used to send a message from one worker to another or to the main execution flow. This code sends a message called message to the w1 worker. The *'worker.receive'* statement is used to receive messages in a worker or the main execution flow.

Synchronization and Joining: Ballerina provides mechanisms to synchronize and join worker execution. The *'worker.wait'* statement is used to synchronize multiple workers, allowing them to wait until a condition is satisfied. The *'worker.join'* statement is used to wait for the completion of a specific worker. This code waits until both the w1 and w2 workers reach a wait point before continuing execution. The *'worker.join'* statement waits for the completion of a specific worker. Workers in Ballerina enable developers to write concurrent and parallel code in a structured manner, easing efficient utilization of system resources. They help handle tasks concurrently, communicate through message passing, synchronize execution when necessary, and enhance performance and scalability in cloud-native applications.

2.2.2.4 Network Interaction

Network interaction plays a fundamental role in modern software systems, allowing applications to communicate and exchange data over various protocols. In Ballerina network interactions are facilitated through the concepts of services and listeners, which enable the implementation of robust and scalable network applications. A ***'service'*** in Ballerina is a logical unit that provides a set of operations or endpoints accessible over the network. It acts as a container for organizing related functionality and serves as the entry point for network interactions. Services encapsulate the behaviour and capabilities of an application or a specific component within the system.

```
service / on new http:Listener(2030) {
    resource isolated function get .() returns Ingredient[]|error {
        return check getAllIngredients();
    }
}
```

Code Example 3 - Example of a 'service' definition

To define a service in Ballerina, you use the ***'service'*** keyword followed by the service name and an optional service-level configuration, as seen in the example above. Inside the service, you can define one or more resource functions that represent the individual endpoints or operations provided by

the service. Each resource function specifies the HTTP method, URL path, and the logic to be executed when the endpoint is invoked.

In the Code Example 3, an HTTP listener is created at the same time the service is defined on the root path, specifying the port number (2030) to listen on. This is a good example of the power of Ballerina in minimizing code to extract largest functionality. The service is associated with this listener using the on keyword. Any incoming HTTP requests to the specified port will be routed to the corresponding resource function within the service.

Listeners and services work together to enable network interaction in Ballerina applications. Listeners supply the entry point for incoming requests, while services define the logic to be executed upon receiving those requests. Through this combination, Ballerina allows developers to easily build and expose network APIs, handle various protocols, and implement scalable and reliable network applications.

Another important consideration about network interaction, when a service receives a request, is the caching of possible results to equal queries. Through this mechanism it is possible to maximize the performance of the service by not requesting other dependencies, remote services or otherwise, and just assume the response is the same. In Ballerina "HTTP caching is enabled by default in the `http:Client`".[10] Therefore the language, in yet another way, simplifies the implementation of API's, requiring only configurations for more specific purposes

2.2.2.5 Testing framework

Ballerina provides comprehensive testing capabilities that enable developers to ensure the quality and reliability of their applications. The language offers built-in testing frameworks and tools that help unit testing, integration testing, and end-to-end testing. The unit testing framework allows developers to write tests for individual functions, services, and modules. It supplies assertion functions that enable developers to verify the expected behaviour of their code. By writing unit tests, developers can confirm the correctness of their code in isolation and identify potential bugs or issues early in the development process.

```
import ballerina/test;

function add(int a, int b) returns int {
    return a + b;
}

function testAdd() {
    int result = add(2, 3);
    test:assertEquals(result, 5, "Addition test failed");
}
```

Code Example 4 - Unit test example [11]

In the example above, we have a function add that performs addition, and a corresponding unit test function *testAdd*. The *test:assertEquals* assertion function is used to verify that the result of the add function is equal to the expected value (5 in this case).

Ballerina's integration testing capabilities allow developers to test the interactions between various components and services within an application. Integration tests help confirm the behaviour of interconnected modules and ensure the smooth functioning of the entire system. Ballerina supplies features such as mocking and stubbing to simulate the behaviour of external dependencies during integration testing. The language also supports end-to-end testing, which involves testing the complete workflow of an application from start to finish. This type of testing ensures that all components, services, and integrations work together correctly. Ballerina provides tools for writing end-to-end tests that simulate real-world scenarios, including interactions with external systems and services.

In addition to the built-in testing frameworks and tools, Ballerina integrates with popular testing frameworks such as JUnit, enabling developers to leverage their existing testing infrastructure and practices.

2.2.3 Infrastructure Features

2.2.3.1 Container Creation

In addition to its core features, Ballerina also includes several tools and development environments that make it easier to write, test, and deploy microservices and other cloud-native applications. For example, Ballerina includes a visual diagramming tool that allows developers to model and visualize their integration flows, as well as a command-line tool for building and deploying Ballerina applications to various cloud platforms.

Such deployment can be done either directly or using containers such as Docker and orchestrated with technologies like Kubernetes. Docker is a popular containerization platform that allows developers to package and deploy applications as lightweight, portable containers that can run anywhere.[14] Ballerina has built-in support for Docker, which means that developers can easily build and deploy Ballerina services as Docker containers. This is done using the 'bal build' command, which creates a Docker image containing the Ballerina service along with its dependencies.

In chapter 4 this is exemplified in further detail on how to properly create and deploy a container imager using Ballerina's integrated tooling system and Docker.

2.2.3.2 Deployment

In addition to Docker, Ballerina also has native support for Kubernetes, which is an open-source container orchestration platform. Kubernetes allows developers to manage and scale containerized applications across a cluster of machines. Ballerina's integration with Kubernetes is done through the Kubernetes extension, which supplies a set of Ballerina language constructs that can be used to create and manage Kubernetes resources, such as pods, services, and deployments.

One of the key features of Ballerina's integration with Kubernetes is its ability to generate Kubernetes artifacts automatically. This means that developers can define their Ballerina services using Ballerina's syntax and Ballerina will automatically generate the corresponding Kubernetes YAML files, which can be used to deploy the services to a Kubernetes cluster. This greatly simplifies the deployment process, as developers do not need to manually create and manage Kubernetes resources.

3 Solution Analysis, Design, and Implementation

3.1 Project Domain

Since this project is meant to be an appreciation of the capabilities of the Ballerina programming language as a tool for microservice development, a simple albeit representative domain was chosen for the purposes of developing the different requirements for the solution. The project will follow the creation of the backend systems for a sandwich shop, which holds the titular item, its ingredients, customers' orders, and the relevant related entities that allow for the complete management of the shop.

The following are the proposed concepts retrieved from the analysis of the requirements from the *'client'* document with the description of the system.

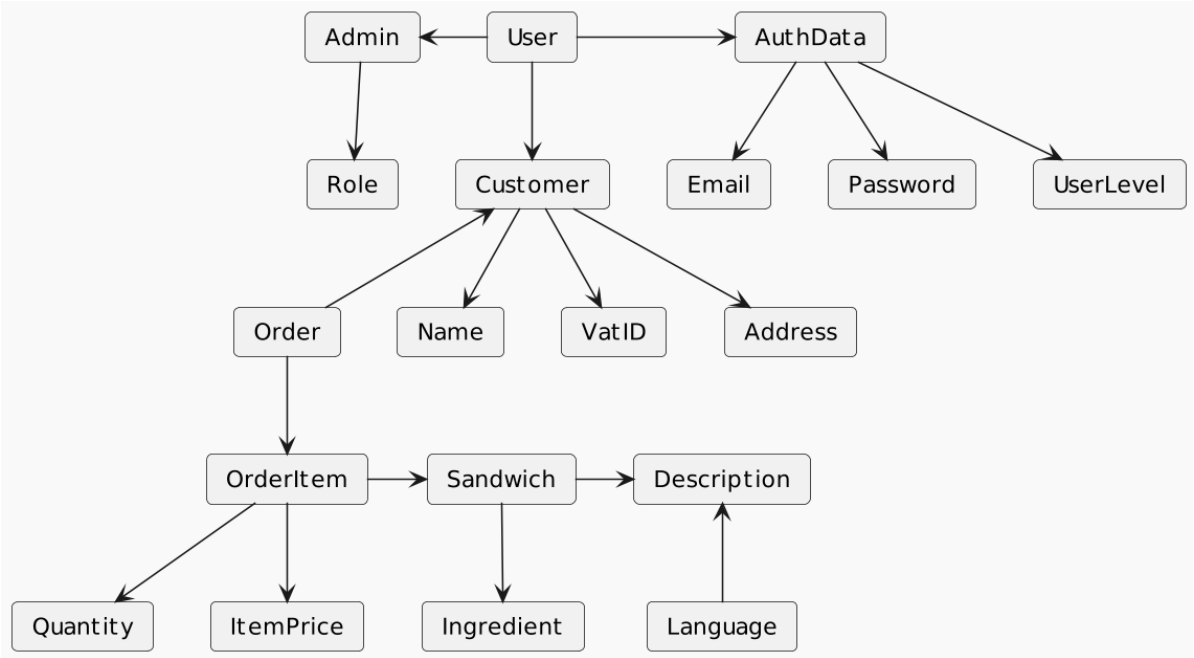


Figure 2 - Domain concepts

Starting on the concept of Sandwich, itself has Ingredients, and Descriptions which are referenced by their language. From there the Orders are composed of OrderItems which allow the registering of each Sandwich type on any given order by quantity and apply an item price. These can be performed by Customers, represented here by simple data, Name, VatID and Address. They are Users who have AuthData, so they can interact with the system. Users are managed by Admins, which are a type of User, meaning that a User can be both an admin and a Customer.

3.2 Functional and Non-Functional requirements

The application and its systems are meant to be developed in a microservice architecture pattern. To that end each of the aggregate roots, in the domain model named as Management Groups, are to be defined in their own isolated service.

Each service must be able to respond to any request withing 3 seconds with a load of 10 concurrent users. The connection to the system must be made through a single-entry point, no user can connect to any specific single service.

Maintainability needs to remain high for the complete system.

3.2.1 Possible use cases

From the proposed domain, 10 use cases were defined as possible initial propositions for the system. As a Customer the client can manage its orders, simple CRUD operations with the *delete* operation referring to the Cancel Order use case. As an Admin, not only can it perform the duties of a Customer, but it can also perform identic CRUD operations on the Sandwich and Ingredient Managements group of use cases. Finally, the System itself can update Orders for the regular use of the application, such that when a Customer needs the status of its order, an updated response is given by the API.

With these use cases an MVP is possible to be achieved for the purposes of showing the capabilities of Ballerina as a language for cloud systems as a complete backend solution and not simply as a middleware piece in the architecture of any given solution.

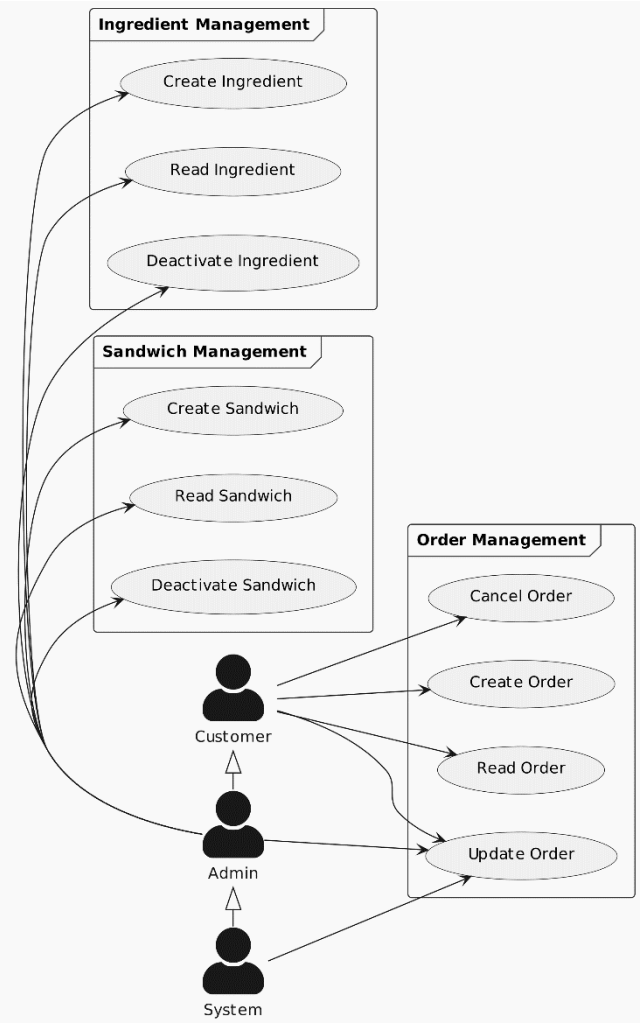


Figure 3 - Proposed use cases

3.3 Solution Design

When creating a web API system for business purposes, one of the first choices to consider is the development method to be employed for building the solution. One option is to adopt a monolithic approach, where the entire software system is a single unit. In this approach, all functionalities and concerns are tightly coupled within the same codebase and deployed as a single application.

Another alternative was to adopt a microservice architecture, which involves breaking down the system into independently deployable services. Each service focuses on a specific concern or functionality, such as user management, payment processing, or data storage. These services can communicate with each other through well-defined APIs, enabling loose coupling and scalability. This approach allows for better isolation and flexibility in scaling individual components based on demand.

Alternatively, a serverless approach can use the capabilities of modern cloud environments. In this model, functionalities are small discrete functions, commonly referred to as “serverless functions” or “function as a service” (FaaS). These functions are deployed and executed in a managed cloud platform, such as AWS Lambda or Azure Functions, which supplies the underlying infrastructure, ensuring scalability, fault tolerance, and automatic resource allocation. This approach abstracts away server and infrastructure management, allowing developers to focus solely on writing the business logic.

By using a Platform as a Service (PaaS) system, developers can deploy and manage their applications without worrying about the underlying infrastructure. The PaaS provider takes care of hardware and networking aspects, allowing developers to focus on writing code and deploying their functionalities. This abstraction simplifies the development and deployment process, enabling faster time-to-market and reducing operational overhead.

In summary, when designing a backend system for a business context, the choice between a monolithic, microservices, or serverless approach depends on factors such as scalability requirements, the complexity of the system, the development team’s ability, and operational constraints. Each possibility offers its benefits and trade-offs, and selecting the most proper one requires careful consideration of the specific project’s needs and goals.

For this project, a microservice architecture was chosen as the ideal structure for the purposes of the work at hand.

3.3.1 Rejected Designs

Rejecting the monolithic approach may be called for when scalability and flexibility are critical factors. Monolithic architectures tend to have a tightly coupled nature, where all functionalities are tightly integrated within a single codebase. As a result, it can be challenging to scale and evolve individual components independently. Monolithic systems can also be unwieldy and difficult to support as they grow in complexity. Therefore, if the project demands the ability to scale different components separately or if it requires a more modular and maintainable architecture, the monolithic approach may not be the best choice.

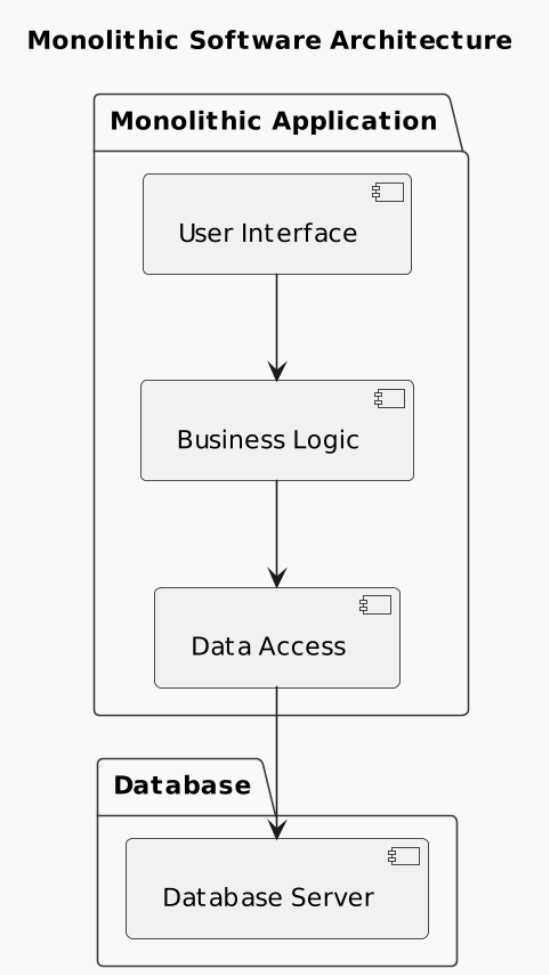


Figure 4 - Monolithic software architecture example

Similarly, rejecting a serverless approach might occur when the nature of the system or the specific requirements do not align well with the serverless paradigm. While serverless architectures offer benefits such as automatic scaling, reduced operational overhead, and pay-per-use pricing, they may not be suitable for all scenarios. Serverless functions are typically stateless and short-lived, making them well-suited for event-driven and ephemeral workloads. However, if the application requires long-running processes, fine-grained control over the underlying infrastructure, or a complex service connectivity, a serverless approach may introduce added complexity or constraints that outweigh the advantages.

Moreover, whilst analysing the final implementation, many considerations were taken in regards of the type of microservice architecture to realize. One type of structure to the services that was considered was one where a single database handled holding all the data for the entire business logic. This was rejected since it would create potential concurrent issues on data writing, and excessive strain on read operations when multiple services were to access the persistence layer.

3.3.2 Proposed solution

In a microservice architecture, where services are decoupled and independently deployable, choosing an API gateway as a solution can supply several benefits (see Figure 6), as discussed previously in chapter 2. Additionally, adopting a single database for each service further enhances the advantages of this architectural approach. As described in Chapter 2.1.1, the system implements the aggregator pattern within the API gateway module, and a dedicated database per service

Below is an example of the flow of communication for the proposed system (Figure 5).

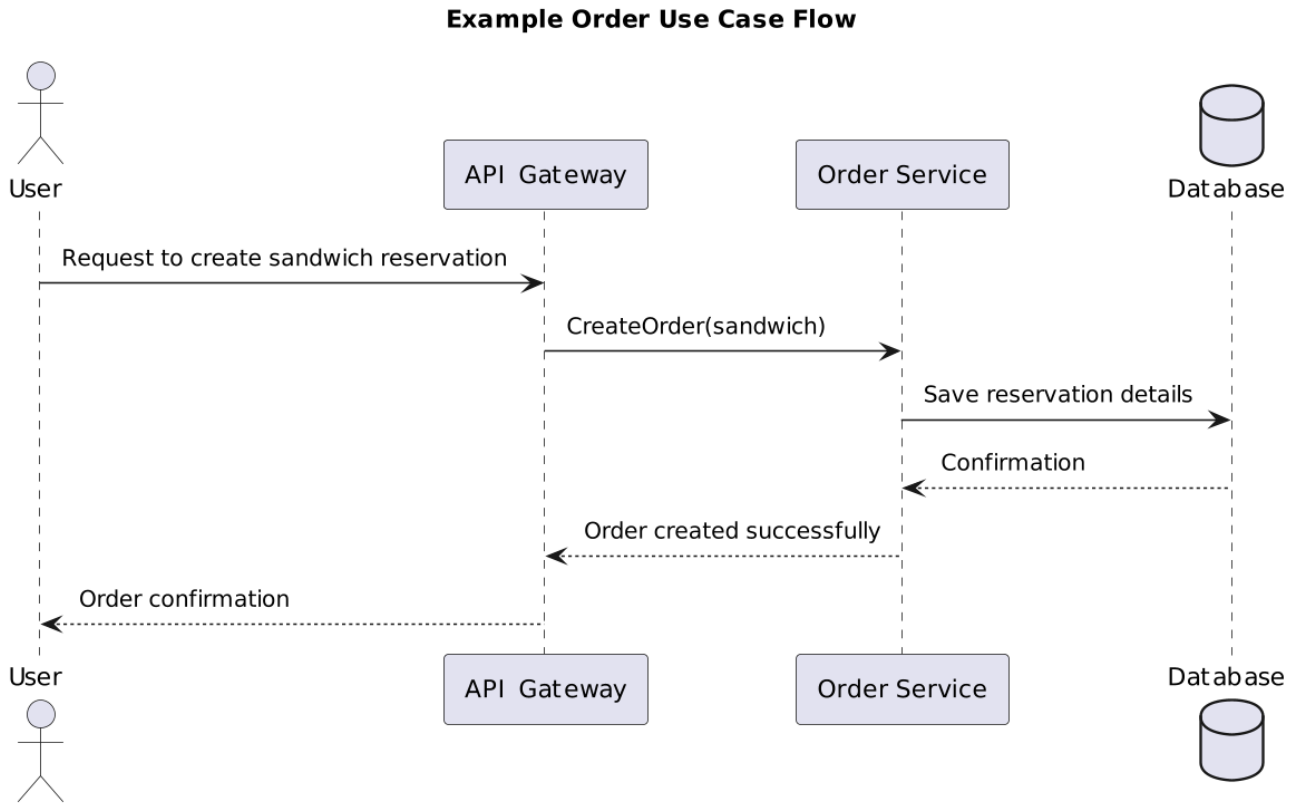


Figure 5 - Example flow for Order creation

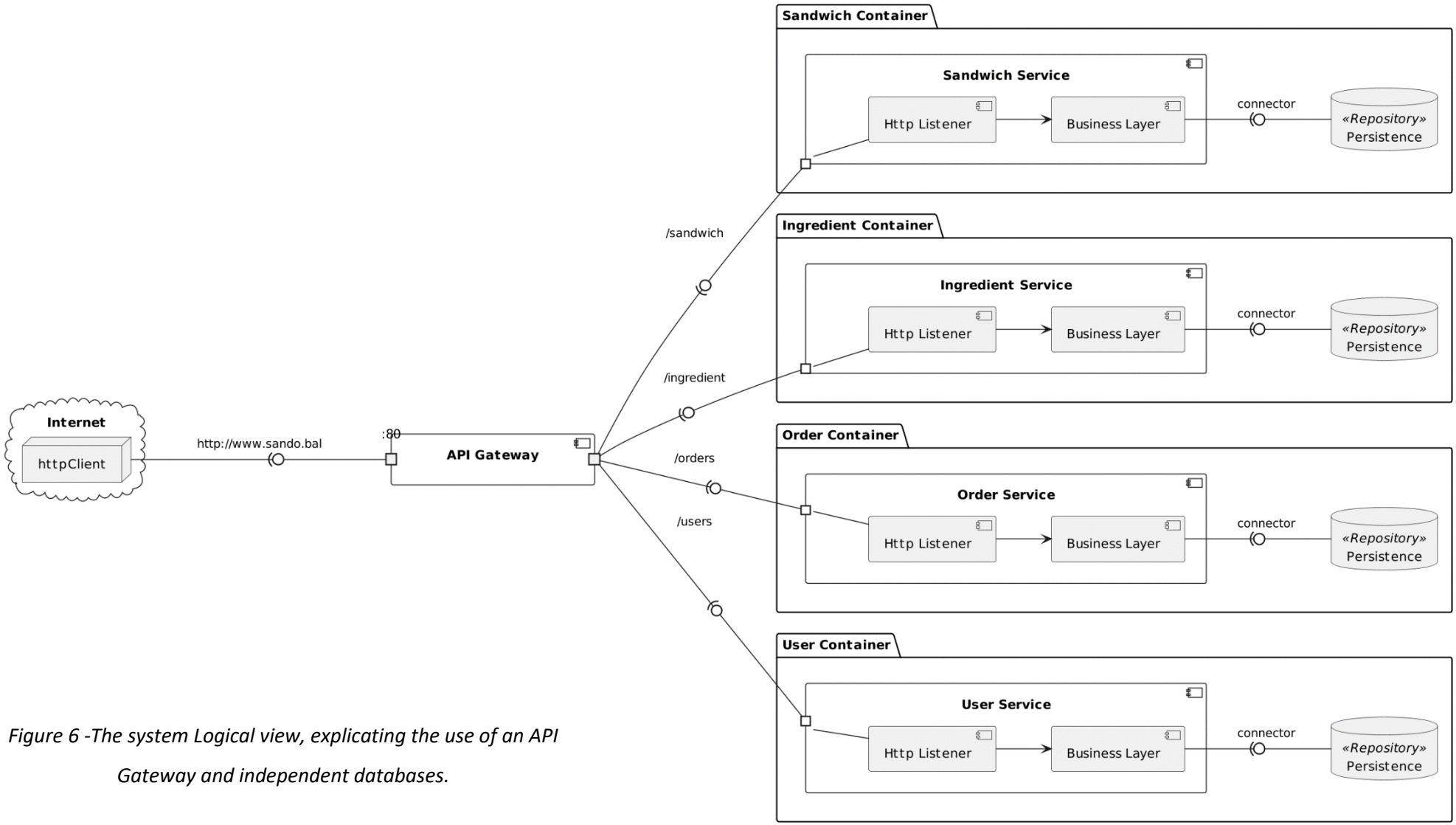
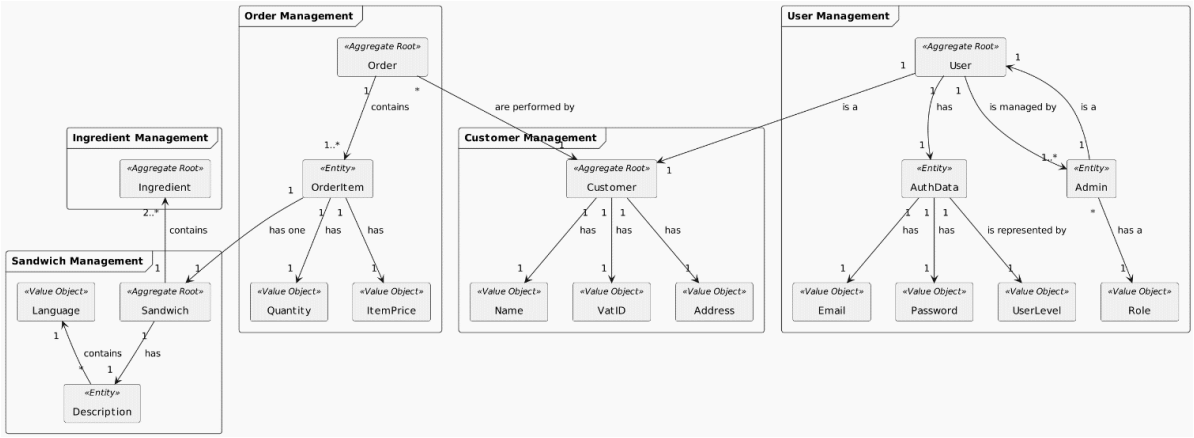


Figure 6 -The system Logical view, explicating the use of an API Gateway and independent databases.

3.4 Implementation

The entirety of this project is developed and stored in a public repository, which will receive further updates to it after the conclusion of this work. [1]

3.4.1 Concretization of Domain



From the domain analysis done previously in the earlier chapter, we can conclude that five Aggregate Roots exists in the project domain. Shown above are the relations between them and their respective Entities. Each of the Management groups are a single service, except for the User Management group, where it is managed in the API Gateway, for authentication and authorization purposes. This last functionality is planned for a future improvement of the project and was not within the scope of this work.

Care was taken in consideration of which concepts would be elevated to Aggregate Roots, in particular User, Customer, Order, Sandwich, and Ingredient. In order, A User refers to a Customer, for it can also be an administrator, which should confer special privileges. An order holds an aggregation of Sandwiches and relates to a Customer. Finally, Sandwich has the information of its ingredients.

3.4.2 Infrastructure as Code

For the implementation and actual deployment of the project, docker containers were used to both isolate and control the separation of concerns of the many services. Ballerina as a language supplies easy means of creating containers in the Docker virtualization system via a simple configuration option in the code. Through this it's possible to create either a Dockerfile or both the file and image

for the container, without any more effort than to run the compilation of code of the project. Therefore, we can use this mechanism, with the aid of a single *docker-compose.yml* file to deploy and instantiate all the necessary and developed services with a persistence layer for each.

To the right there's an example of the definition of the sandwich micro service as a dockerized image and it's corresponding database instance.

```
sando-ms:
  container_name: sandwich-api
  image: sandobal-api-sandwich:latest
  build:
    context: api.sandwich/target/docker/sandobal_sandwich_api/
    dockerfile: Dockerfile
  networks:
    - sandwichnet
    - servicenet
  restart: always
  links:
    - sando-db
  depends_on:
    - sando-db
  volumes:
    - ./api.sandwich/Config.toml:/home/ballerina/Config.toml
sando-db:
  container_name: sandwich-data
  image: postgres:latest
  restart: unless-stopped
  networks:
    - sandwichnet
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_DB=sandwich
  volumes:
    - sando-db:/var/lib/postgresql/data
```

Of relevance it's observable that the container exposes no ports, but instead relies on pre-defined networks on the docker system, *sandwichnet* and *servicenet*, for the microservice. The first one is the network that allows the connection between the service and its database, the second is the network that holds and exposes all services and the gateway, for communication between them. Using the compose file it's possible to easily deploy all the infrastructure in a CI, Continuous Integration, like fashion.

This makes it easy to run integration and load testing on the entire system.

3.4.3 Architecture Implementation

3.4.3.1 API Gateway Implementation

The API Gateway serves the various use cases for the project as http resources to consume. Since caching is implemented by default in the *http:Client* class, no other considerations were taken for cache usage thus allowing to test the Ballerina default implementation.

3.4.3.1.1 Entry Point

```
import ballerina/http;

listener http:Listener controllerListener = new (9090);
final Orchestrator orca = check new ();

service /sandwich on controllerListener {

    resource function post create(SandwichDTO sand) returns int|error {
        return check orca.createSandwich(sand);
    }
}
```

Code Example 5 - API Gateway entry point

The entry point of the gateway module starts by defining a Listener object on the port 9090, this will make it so that all HTTP requests are only accepted in this manner.

Afterwards the service, defined by the path '*/sandwich*' is attached to the listener object previously defined. Within the service, resource functions are created, and these consumable resources are exposed through the listener object, and are defined through the keywords '*get, put, post, delete*' and the method name.

With this definition of the service the routing is automatically defined and set, with the above example corresponding to a service that accepts the *SandwichDTO* object in an HTTP POST method, in the path *http://localhost:9090/sandwich/create*.

3.4.3.1.2 Orchestrator

```
# The Orchestrator class of the API Gateway, which also implements the aggregator pattern for DTO consolidation
public isolated class Orchestrator {

    private final IngredientClient ingredientEndpoint;
    private final SandwichClient sandwichEndpoint;

    public isolated function init() returns error? {
        self.ingredientEndpoint = check new ({}, ingredientApiUrl);
        self.sandwichEndpoint = check new ({}, sandwichApiUrl);
    }

    public isolated function createSandwich(SandwichDTO sandwich) returns int|error {

        CreateSandwichDTO command = {
            selling_price: sandwich.Price,
            designation: sandwich.Name,
            descriptions: sandwich.Descriptions
        };

        command.ingredients =
        from Ingredient ing in (
            from IngredientDTO dto in sandwich.IngredientsList
                  select check self.getIngredient(dto.name)
            )
        select ing.ingredient_id;

        var result = check self.sandwichEndpoint->/create.post(command);
        return result;

    }
}
```

Code Example 6 - Orchestrator Class Excerpt

The example above is a section of code of the Orchestrator class where the communication with the backend services is done for the execution of the requested use case. It shows the usage of the Sandwich and Ingredient client Classes, which are implmentations in the API Gateway of the respective service interface, to create a new sandwich.

Of note we have the section to the right, where the Orchestrator class creates a list of Ingredients reference by way of an SQL like syntax, where a nested selection of data, coming from the *self.getIngredient()* function, is collated into a queryable iteratable construct with a defined type, and is subsequently selected for the relevant data, in this case *ingredient_id*.

```
command.ingredients =
from Ingredient ing in (
            from IngredientDTO dto in sandwich.IngredientsList
                  select check self.getIngredient(dto.name)
            )
select ing.ingredient_id;
```

This is saved in the property *ingredients* of the *command* object, which then is used to consume the *create* resource in the *sandwichClient*.

```
        var result = check self.sandwichEndpoint->/create.post(command);
```

The line above reads that, the variable *result* shall be assigned the result of the request to the resource *'/create'* through an *HTTP POST* method with the body content containing the *command*

object in the instantiated *sandwichClient*, whilst being flagged for a possible error return, through the *check* keyword which will halt immediately the execution of the method.

3.4.4 Service Implementation

For the purposes of comparison of Ballerina with another common-use and industry relevant solution, a comparison with C# was implemented. The sandwich service was chosen, because it's where creation and manipulation of core data to the system is done, and responsiveness is mandatory for the better use of the API.

3.4.4.1 Ballerina implementation

The implementation of the service in Ballerina follows a similar syntax to that of the API gateway.

```
service / on new http:Listener(2020) {
        isolated resource function post create(CreateSandwichDTO sandwich) returns int|error{
        return check createSandwich(sandwich);
    }
}
```

Code Example 7 - Sandwich Service entry point excerpt

From here the code calls the relevant backend function, in this case *createSandwich()*.

```
configurable string host = "";
configurable string username = "";
configurable string password = "";
configurable string database = "";
configurable int port = 5432;

//DEBUG CONFIG
final postgresql:Client dbClient = check new (host, username, password, database, port);

isolated function createSandwich(CreateSandwichDTO command) returns int|error {

    if getSandwich(command.designation) is Sandwich {
        io:println("Sandwich already exists");
        return -1;
    }

    //CREATE SANDWICH
    sql:ParameterizedQuery createSandwichQuery = `INSERT INTO sandwich ( designation, selling_price)
                                            VALUES ( ${command.designation}, ${command.selling_price})`;

    var result = check dbClient->execute(createSandwichQuery);

    var createdId = result.lastInsertId;

    //CREATE INGREDIENT
    from int ingredient_id in command.ingredients
    do {
        if existsIngredient(ingredient_id) {
            sql:ParameterizedQuery createSandIngQuery = `INSERT INTO SandwichIngredients (sandwich_id, ingredient_id)
                                            VALUES (${createdId}, ${ingredient_id})`;

            sql:ExecutionResult _ = check dbClient->execute(createSandIngQuery);
        } else {
            sql:ParameterizedQuery deleteSandIngQuery = `DELETE FROM sandwich WHERE sandwich_id = ${createdId};`;
            sql:ExecutionResult _ = check dbClient->execute(deleteSandIngQuery);
            return -1;
        }
    };

    //CREATE DESCRIPTIONS
    from Description desc in command.descriptions
    do {
        sql:ParameterizedQuery createSandDescQuery = `INSERT INTO SandwichDescriptions (sandwich_id, content, language)
                                            VALUES (${createdId}, ${desc.content}, ${desc.language})`;

        sql:ExecutionResult _ = check dbClient->execute(createSandDescQuery);
    };

    return <int>result.affectedRowCount;
}
```

Code Example 8 - Create Sandwich method implemented in Ballerina

The code above is the implementation of the creation of a sandwich entity in the backend of the module. It starts by first creating a database access client, in this case for the postgresql technology, using a set of *configurable* variables. These are values that Ballerina automatically looks for in a configuration file, within the project folder, called *Config.toml*.

The method starts by searching the input parameter entity in the existing database, returning at once if it already does. Then, using precompiled *sql:ParametrizedQuery* objects, the creation of the various

data entry points is done using the *execute* remote method. This calls an external library, *ballerina/postgresql,* to perform the operation.

A point of improvement in this code is the use of *workers* and *transactions*, base types of Ballerina, to further optimize the implementation of the system.

Other resources on the service are implemented in a similar fashion and are available at the repository of this project.[1]

3.4.4.2 C# implementation

The C# implementation of the service was done with the .NET 7.0 version of the language. It follows the same logic of the Ballerina implementation of the same service, and it uses the EntityFrameworkCore ORM for more easily manage Entities and the database connection. For data manipulation the library LINQ and its query call structure was used to simplify the filtering of data.

```
[ApiController]
[Route("")]
public class SandwichController : ControllerBase
{
    [HttpPost("create")]
    public async Task<IActionResult> createSandwich([FromBody] CreateSandwichDTO dto)
    {
        Sandwich newsand = new()
        {
            Designation = dto.designation,
            IsActive = true,
            Sandwichdescriptions = dto.descriptions.Select(d =>
                { return new Sandwichdescription { Content = d.content, Language = d.language }; }).ToList(),
            SellingPrice = dto.selling_price,
            Sandwichingredients = dto.ingredients.Select(i =>
                { return new Sandwichingredient { IngredientId = i }; }).ToList()
        };
        await data.Sandwiches.AddAsync(newsand);
        if ((await data.SaveChangesAsync()) > 0)
        {
            return Created("", $"Sandwich {newsand.Designation} id: {newsand.SandwichId} was created.");
        }
        else
        {
            return BadRequest("Sandwich not Created");
        }
    }
}
```

Code Example 9 - C# implementation of the Sandwich Service create method

4 Results

4.1 Performance Tests

4.1.1 Methodology

For the purposes of testing the system, three main tests were executed per what was described on Chapter 2.1.2. For the comparison between C# and Ballerina a simultaneous execution of testing was performed, both service instances were loaded with requests at the same time, to remove the variability of possible CPU spikes adding uncontrolled data between runs. This way any variability will occur in both concurrent systems in equal measure.

All tests were designed per the non-functional requirement of any response to be less than 3 seconds whilst 10 concurrent users executed requests. To achieve this the throughput of requests was kept above 10 simultaneous, and response time is the relevant variable under analysis.

The testing framework was the software JMeter, and the test definitions is also available in the main repository for the project. All measured values in the Average, Min and Max columns are in milliseconds, and in Throughput it refers to average requests per second.

For the comparison between C# and Ballerina, three groups of tests were designed. A Single HTTP GET method in each language, a Double HTTP Get method, and a Single POST method. The GET methods are simple execution requests of a single entity by a given token, id for the Single GET Test, and id and name for the Double GET. The reason for this design is that this way the same database is accessed and so it's possible to see how the two languages deal with concurrent access to the same data, and how the caching mechanism works.

The cache was not changed or disabled in either case, as it is an industry standard to use such a system in distributed systems, and it's functioning a good test to perform as different languages implement it differently.

For the single POST method, the creation of a new entity, a Sandwich record, each time a request is made was determined to be sufficient to be able to perform a comparison between the two languages. Both systems work at the same time to create the new entity, and access the same database, making queries to confirm other related data before committing the creation of the entity. This is then a good test of for comparing this particular action between the languages.

All tests are performed at the same time to equalize the overhead that they and other background processes might cause on the executing machine's CPU. This way the resulting response times are representative of the computation performed.

4.1.2 Ballerina System Results

4.1.2.1 Load Testing

For Load testing a startup time of 20 seconds up to 50 possible simultaneous users was executed for 50 seconds and an end time of 10 seconds, to a total of 90 seconds.

Figure 7 - Plan for the system's Load testing

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP GET - All Ingredients	1637	8	0	26	1.73	0.000%	18.82042	5.44	2.39	296.0
HTTP GET - Sandwich by Name	1607	12	0	24	2.24	0.000%	18.54373	6.90	2.50	381.0
HTTP POST - Create Reservation	1607	32	0	95	7.48	0.000%	18.54095	3.77	5.90	208.0
TOTAL	4851	17	0	95	11.84	0.000%	54.67332	15.75	10.55	295.0

Table 1 - Load testing results

4.1.2.2 Soak Testing

For the Soak testing a load of at least 50 users was held for a little over 8 minutes, 500 seconds.

Figure 8 - Plan for the system's Soak testing

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP GET - All Ingredients	12321	7	0	274	3.14	0.000%	22.43735	6.49	2.85	296.0
HTTP GET - Sandwich by Name	12295	10	0	260	5.13	0.000%	22.40551	8.34	3.02	381.0
HTTP POST - Create Reservation	12295	27	0	275	8.12	0.000%	22.40457	4.56	7.13	208.5
TOTAL	36911	15	0	275	10.79	0.000%	67.21736	19.38	12.99	295.2

Table 2 - Soak testing results

4.1.2.3 Spike testing

For the spike test a maximum of 1000 users were created in two separate spikes of requests for concurrent stress of the system, while 50 other users maintained a constant level of load on the system.

Figure 9 - Plan for the system's Spike testing

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP GET - All Ingredients	1134	94	0	448	84.87	0.000%	53.06008	15.34	6.74	296.0
HTTP GET - Sandwich by Name	338	92	0	520	116.50	0.000%	16.34825	6.08	2.20	381.0
HTTP POST - Create Reservation	274	207	0	915	236.34	0.000%	13.24311	2.70	4.22	209.0
TOTAL	1746	111	0	915	133.31	0.000%	81.69568	23.84	12.95	298.8

Table 3 - Spike test results

4.1.3 C# vs Ballerina Comparison

For the comparison between Ballerina and C# the same Load profile was used for all requests, meaning each request loaded 50 concurrent users for requests creation and held the stress test for 30 seconds, with a startup time of 20 seconds and a cooldown time of 5 seconds. For each test, all the requests were ran concurrently as seen in the results tables below and ran for 60 seconds total time each.

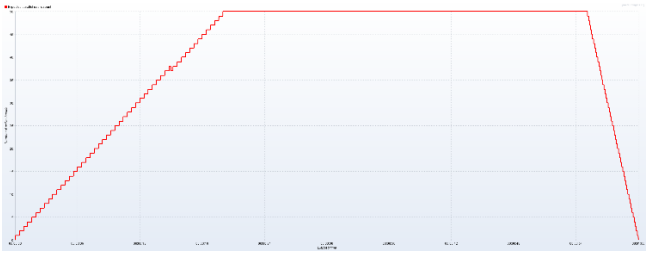


Figure 10 - Plan for the comparison between C# and Ballerina testing

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
BAL GET - Sandwich by Id	1490	5	0	61	3.00	0.000%	25.46312	9.77	3.01	393.0
C# GET - Sandwich by Id	1467	3	0	37	1.64	0.000%	25.71293	7.26	3.04	289.0
TOTAL	2957	4	0	61	2.83	0.000%	50.53319	16.85	5.97	341.4

Table 4 - C# v Ballerina Single GET method results.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
BAL GET - Sandwich by Id	578	5	0	9	0.72	0.000%	9.83478	3.77	1.16	393.0
BAL GET - Sandwich by Name	566	4	0	8	0.68	0.000%	9.88733	3.81	1.29	395.0
C# GET - Sandwich by Id	549	3	0	5	0.55	0.000%	9.63276	2.72	1.14	289.0
C# GET - Sandwich by Name	540	2	0	11	0.65	0.000%	9.38918	2.67	1.23	291.0
TOTAL	2233	3	0	11	1.39	0.000%	37.99493	12.74	4.73	343.3

Table 5 - C# v Ballerina Double GET method results.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
BAL POST - Create Sandwich	1462	7	0	50	2.89	0.000%	24.77588	3.19	10.91	132.0
C# POST - Create Sandwich	1444	9	0	57	4.33	0.000%	24.54739	5.27	10.81	220.0
TOTAL	2906	8	0	57	3.80	0.000%	49.20337	8.44	21.67	175.7

Table 6 - C# v Ballerina Single POST method results

4.1.4 Discussion

From the results of the testing on the system, it's easy to observe that Ballerina as a language is perfectly capable of being a tool for developing microservices. The performance under load does not show any overhead from lack of optimization for the widespread use case of being used for web services. Also as predicted before the execution of tests, CPU availability variability was a factor to consider on the execution plan for the tests. Annex II displays an unexpected momentary increase in response time, which can be attributed to a CPU spike that delayed all the processing responses. These increased all at the same time and proportionately in regards on each other.

All tests executed two HTTP GET methods and one HTTP POST method concurrently. The POST method, Create Reservation, being the most complex one in terms of computational effort in the system, could respond well within the project's limits. As it's observed in the Spike test, the maximum response time for the POST method was 915 milliseconds, and the highest average was slightly below 750 milliseconds. It shows a very consistent behaviour throughout the execution of the tests. This can be clearly seen in the graph in Annex I, II and III.

As per the non-functional requirements all requests were with over 10 concurrent users and under 3 seconds per response time, which can be seen in all test result having above 50 concurrent user requests a second with response times below 1 second.

In the comparison tests between C# and Ballerina we can see a clear difference in performance, the ANNEXES IV, V and VI help with this visualisation. This can be due to many factors, like implementation optimization, language optimization due to existing for longer. The difference is Ballerina having almost double the average response time of C# in most tests, something visible in the tables the sections above. Whilst this is true it is also of note that the differences are below 3 milliseconds for all the HTTP methods regardless of them being GET or POST methods. This shows that Ballerina might have a bit of an overhead of computation when executing requests, that C# does not. This difference is also far below 1% of the requirement for this project of 3 seconds of response time.

Thus, we can easily conclude that the performance between the two languages in the tests performed is identical.

4.2 Ballerina as an industry solution

4.2.1 Implementation

Ballerina is designed to enable the development of complex systems while minimizing development time and code lines. Its goal is to supply a robust deployment process right from the start.

With Ballerina, developers can use its built-in support for distributed systems, service orchestration, and network protocols to easily design and implement complex architectures. The language offers a rich set of features, including native support for microservices, built-in connectors for various protocols, and seamless integration with cloud platforms and APIs.

Furthermore, Ballerina includes extensive tooling and observability capabilities that enhance the deployment experience. It offers built-in testing, debugging, and monitoring features, making it easier to confirm and troubleshoot complex systems during development and deployment.

By emphasizing deployment and operational aspects from the beginning, Ballerina aims to minimize the time and effort needed to build resilient and scalable systems. It supplies a comprehensive framework for handling service discovery, load balancing, circuit-breaking, and other important aspects of distributed systems, which are crucial for reliable deployments.

4.2.2 Complexity

Ballerina is designed to provide simplicity and ease of use in developing both server-side and client-side applications. The language follows a “minimalistic” approach, aiming to reduce complexity and boilerplate code while keeping expressiveness and power. Ballerina achieves this simplicity through its concise syntax and a small set of keywords that cover a wide range of functionality. It provides built-in support for handling network protocols, concurrency, and distributed systems, allowing developers to focus more on the business logic rather than dealing with low-level implementation details.

Comparing Ballerina to C#, a widely used language used in this project for the purposes of comparison, we can see differences in terms of complexity. C# is a feature-rich language with a larger set of keywords and a comprehensive class library. While it provides extensive capabilities for application development, it also requires developers to have a deeper understanding of its various constructs and concepts.

In contrast, Ballerina abstracts away many complexities associated with building distributed systems by supplying higher-level constructs, such as service-oriented programming, message-passing

concurrency, and built-in support for network protocols like HTTP and gRPC. This simplifies the development process and enables developers to quickly create server-side and client-side applications with fewer lines of code.

Ultimately, the choice between Ballerina and other existing web capable languages depends on the specific requirements of the project, the expertise of the development team, and the ecosystem surrounding the target platform.

4.2.3 Maintainability

Due to Ballerina's concise and expressive syntax, each line of code typically stands for a clear and well-defined functionality. This can lead to lower cyclomatic complexity compared to languages that rely on more verbose syntax or annotations. By providing a focused set of keywords and built-in abstractions for common tasks in distributed systems, Ballerina allows developers to express their intent in a concise and readable manner. This often results in more straightforward code that is easier to understand, analyse, and reason about.

As a new modern language Ballerina has some disadvantages compared to its competitors, namely C# and JAVA, as a broad industry solution. In particular, because of its recency, many breaking changes are present in its frequent updates. [13] This does not allow for a consistent and long-lasting development cycle for a team, as many new features and, more importantly, bugfixes are not available if version restrictions to the language level are applied. This is common in many situation, but for the future of a new emerging tool, like Ballerina, being up to date with the language version is necessary for stability and performance. Thus, this does lead to extra tech debt, and extra dev time that might now have been initially predicted.

Ballerina, as a programming language, excels in providing concise and expressive ways to define complex resources such as services and remote functions with minimal source code. This characteristic offers several advantages, including reducing the potential points of failure and abstracting away unnecessary boilerplate code. However, it is worth noting that this conciseness and expressiveness can potentially increase the cyclomatic complexity of the language. The concise nature of Ballerina allows developers to define services and remote functions with remarkable brevity. By using intuitive language constructs and built-in abstractions, Ballerina enables the implementation of complex functionalities in a succinct manner. This concise coding style helps improve code readability and maintainability, as it reduces verbosity and allows developers to focus on the core logic of their applications. The reduction in the number of lines of code can have a positive

impact on the overall reliability of the software. With fewer lines of code, there are fewer opportunities for bugs or errors to creep in. By minimizing the surface area for potential failures, Ballerina promotes code that is easier to understand, debug, and support. Additionally, the reduced code size can enhance developer productivity by speeding up development cycles and reducing the cognitive load associated with large codebases.

On the other hand, the conciseness of Ballerina can result in increased cyclomatic complexity. Cyclomatic complexity measures the number of possible execution paths through a program, and a higher complexity value implies a greater number of potential decision points. While Ballerina's conciseness helps streamline code, it may also lead to more intricate logic structures and nested conditions, potentially increasing the cyclomatic complexity. A higher cyclomatic complexity can make code comprehension and debugging more challenging, as it introduces a greater number of possible execution paths and decision points. It is important for developers to carefully manage and structure their code to avoid excessive complexity. Techniques such as refactoring, modularization, and code reviews can help mitigate the negative effects of increased complexity and keep code quality.

```
if config.proxy is http:ProxyConfig {
        httpClientConfig.proxy = check config.proxy.ensureType(http:ProxyConfig);
}
```

Code Example 10 - Example of namespace in external lib object

When several libraries are in use, with non-colliding names and given the naming of the objects is precise, the excess characters when trying to read unfamiliar code might create excessive reading difficulty for the developer.

5 Conclusion

To finalize it's easy to conclude that, whilst Ballerina is not the optimal solution for heavy computational problems, like machine learning, CFD or BigData manipulation, it is certainly an optimal solution for distributed systems dealing with delivering data and acting as a middleware between client systems and internal backend architectures, be them other computational or persistence resources.

The implementation style, while it has its particularities, is trivial to the used programmer, and provides great abstractions for code that would otherwise be extremely repetitive, commonly known as boilerplate code, a frequent problem in other languages.

The performance in perfectly on par with the most common languages used in the industry. It is a language that performs its use case exactly as intended, has many improvements planned over time, and has been growing in adoption in the latest years.

6 Bibliography

[1] M. Pereira, https://github.com/migueljrpereira/sando-bal (Accessed: 26 February 2023)

[2] A. Maiti, Api Gateway Desing Pattern in microservices, https://www.c-sharpcorner.com/article/microservices-design-using-gateway-pattern/ (accessed Jun. 4, 2023).

[3] U. Hafeez, "Microservice architecture, its design patterns and considerations," C# Corner, https://www.c-sharpcorner.com/article/microservice-architecture-its-design-patterns-and-considerations/ (accessed May 25, 2023).

[4] C. Richardson, "Microservice Architecture pattern," microservices.io, 2023. https://microservices.io/patterns/microservices.html (Accessed May 25, 2023).

[5] A. Almeida, *Ballerina: Integration programming language*, *Medium*. Available at: https://medium.com/ballerina-techblog/ballerina-integration-programming-language-5d8e1b52e582 (Accessed: 01 July 2023).

[6] S. Jayasoma, *Introduction to the Ballerina 0.970 release* https://blog.ballerina.io/posts/2018-06-04-introduction-to-ballerina-0.970/ (Accessed: 01 July 2023)

[7] Ballerina-Platform, *examples/asynchronize-message-passing/asynchronize_message_passing.bal*. Available at: https://github.com/ballerina-platform/ballerina-distribution/blob/master/examples/asynchronize-message-passing/asynchronize_message_passing.bal (Accessed 20 May 2023)

[8] T. Jewell, "Ballerina microservices programming language: Introducing the latest release and 'Ballerina central,'" InfoQ, https://www.infoq.com/articles/ballerina-microservices-language-part-1/ (accessed Jul. 2, 2023).

[9] W. LLC, "Ballerina - check expression," Ballerina Home, https://ballerina.io/learn/by-example/check-expression/ (accessed Jul. 2, 2023).

[10] Ballerina-Platform, "Ballerina-distribution/examples/HTTP-caching-client/http_caching_client.MD at master · ballerina-platform/ballerina-distribution," GitHub, https://github.com/ballerina-platform/ballerina-distribution/blob/master/examples/http-caching-client/http_caching_client.md (accessed Jul. 2, 2023).

[11] W. LLC, "Ballerina - test a simple function," Ballerina Home, https://ballerina.io/learn/test-ballerina-code/test-a-simple-function/ (accessed Jul. 2, 2023).

[12] “What is an API gateway? A quick learn guide: Nginx Learning,” NGINX, https://www.nginx.com/learn/api-gateway/ (accessed Jul. 2, 2023).

[13] W. LLC, “Ballerina - 2201.6.0 (Swan lake),” Ballerina Home, https://ballerina.io/downloads/swan-lake-release-notes/swan-lake-2201.6.0 (accessed Jul. 6, 2023).

[14] B. Borisov, “How to Install Docker on Linux Mint 21: A Step-by-Step Guide,” *Linuxiac*, Mar. 2023, [Online]. Available: https://linuxiac.com/how-to-install-docker-on-linux-mint-21/

ANNEX I

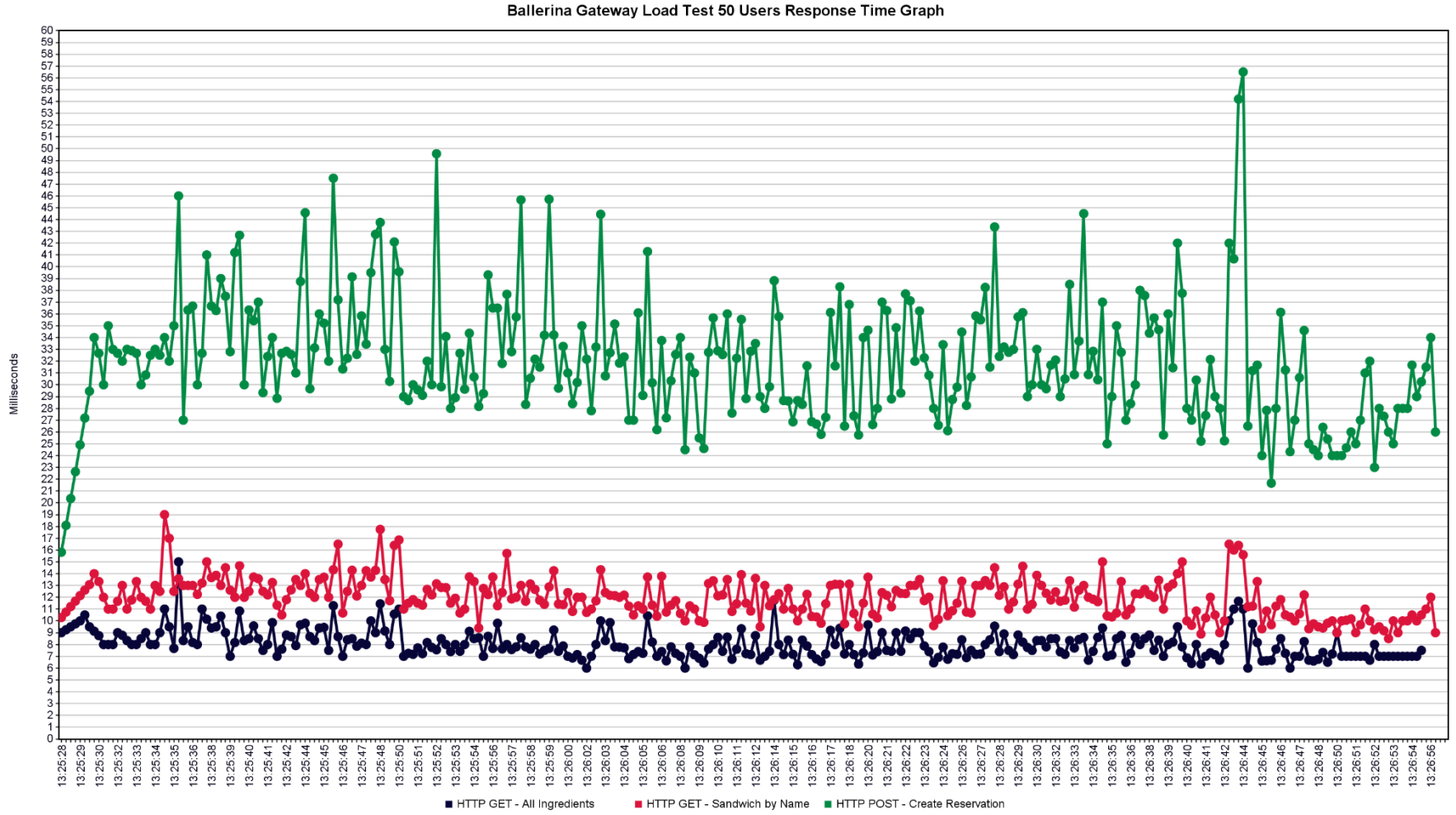


Figure 11 - System Load Test Response time Graph

ANNEX II

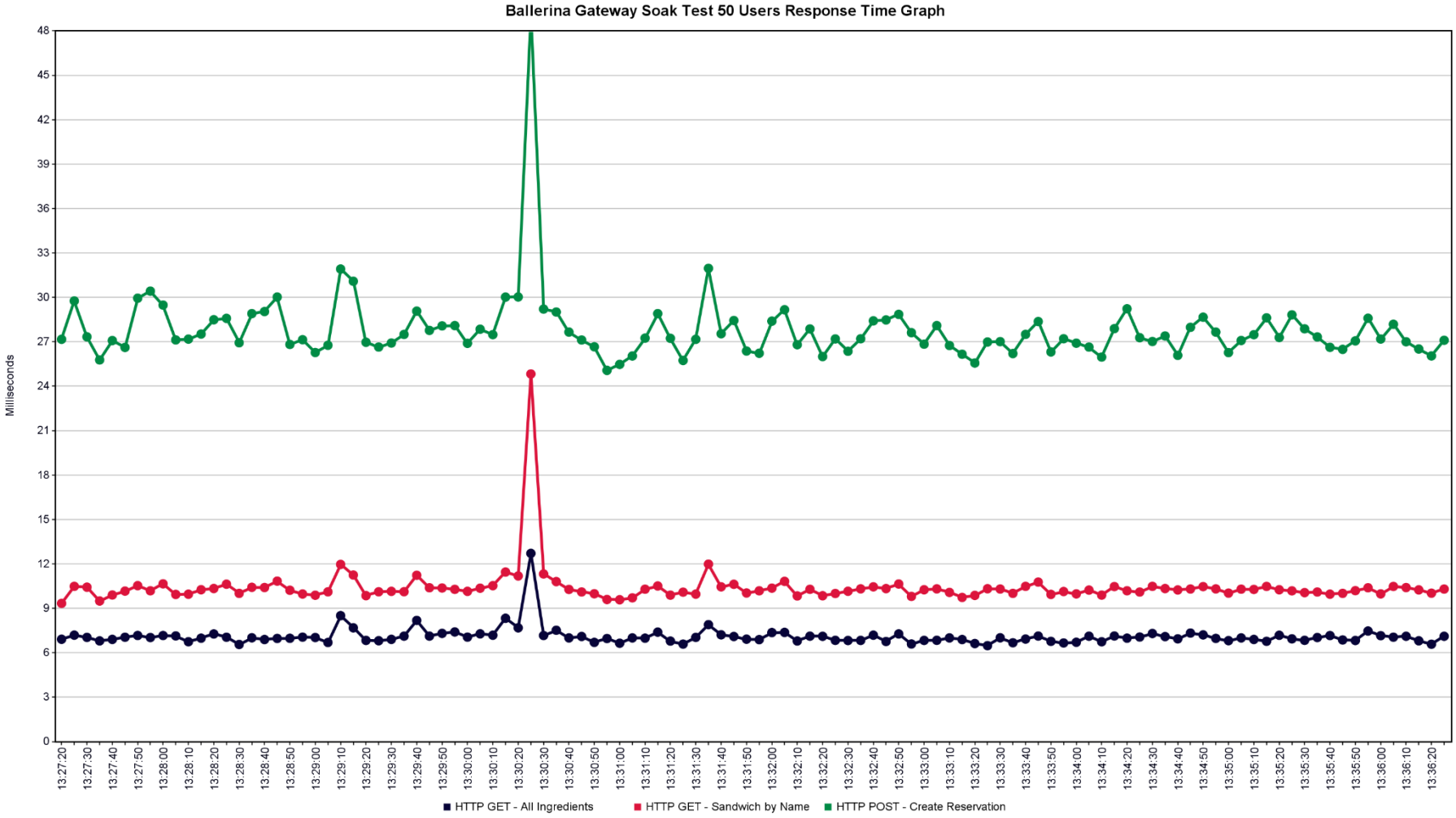


Figure 12 - System Soak Testing result graph

ANNEX III

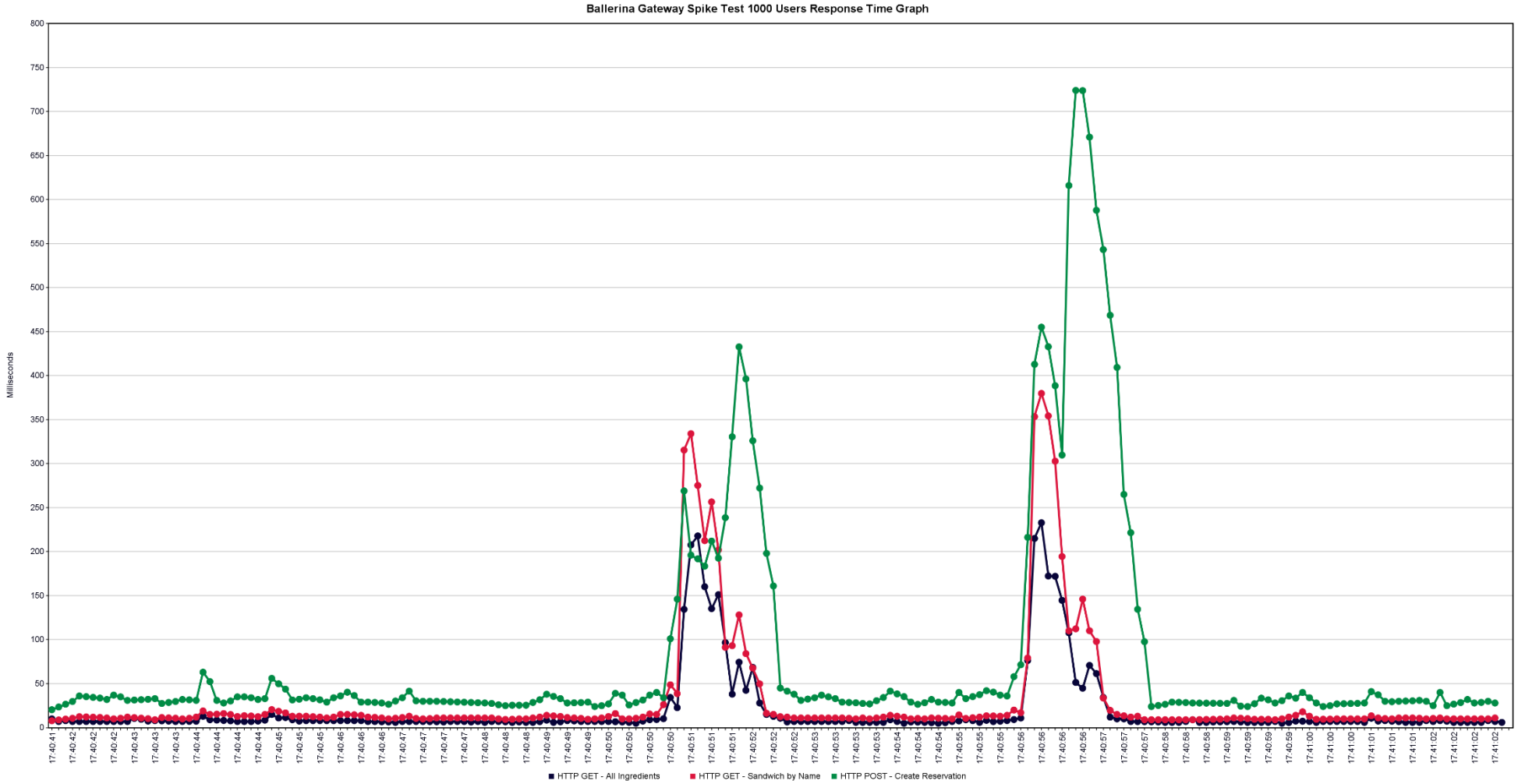


Figure 13 - System Spike stress test results graph

ANNEX IV

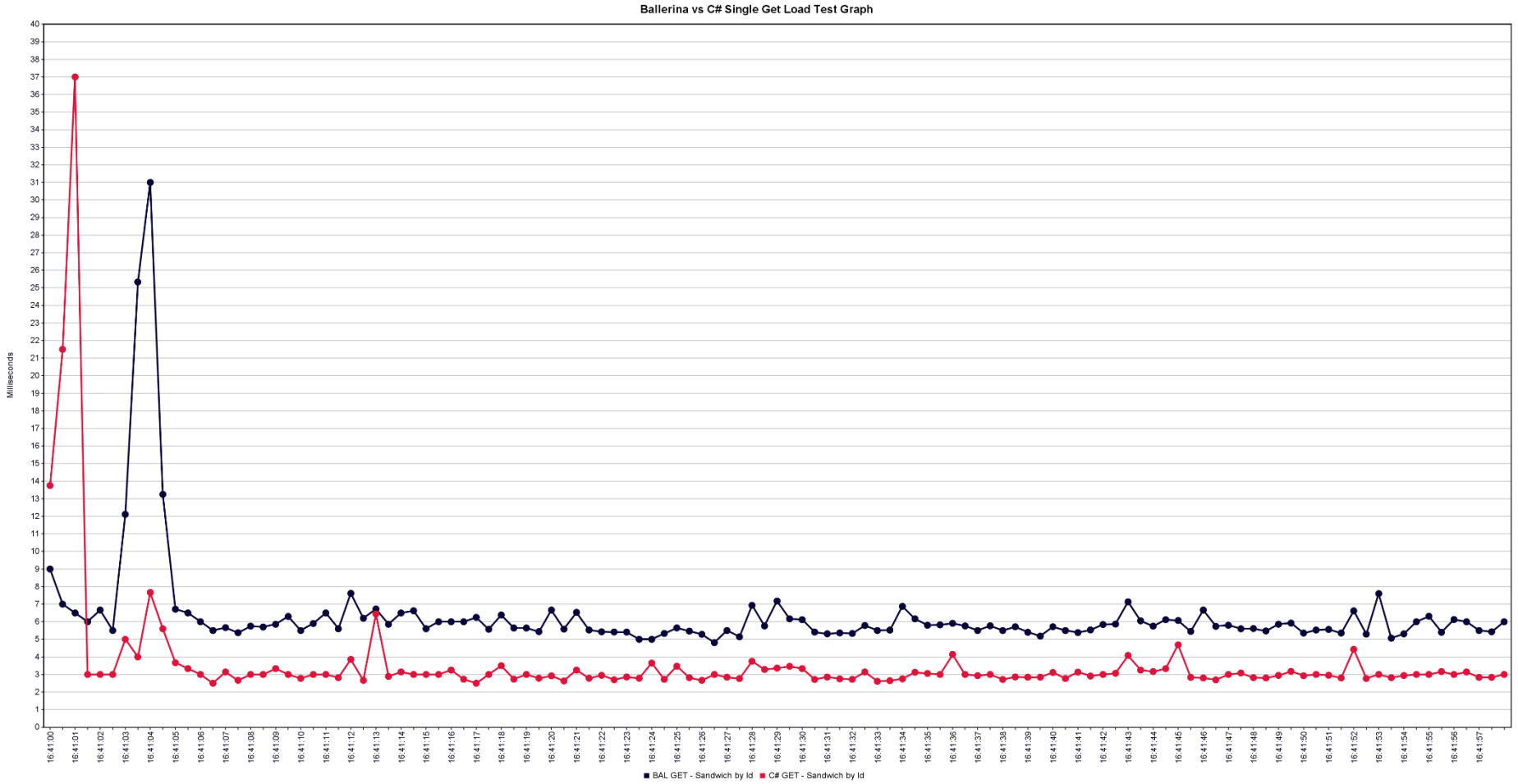


Figure 14 - C# v Ballerina Single GET method response time graph.

ANNEX V

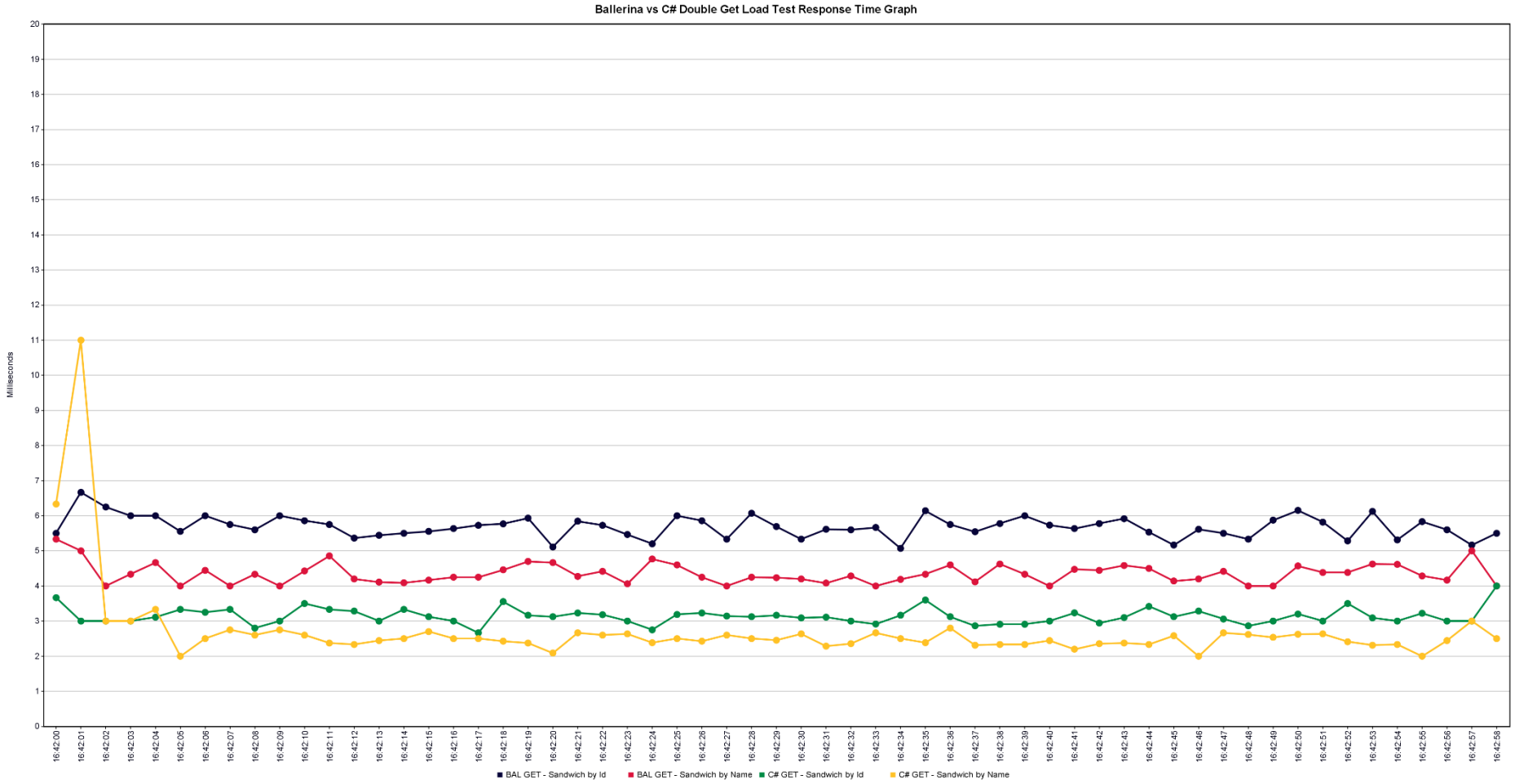


Figure 15 - C# v Ballerina Double GET method response time graph.

ANNEX VI

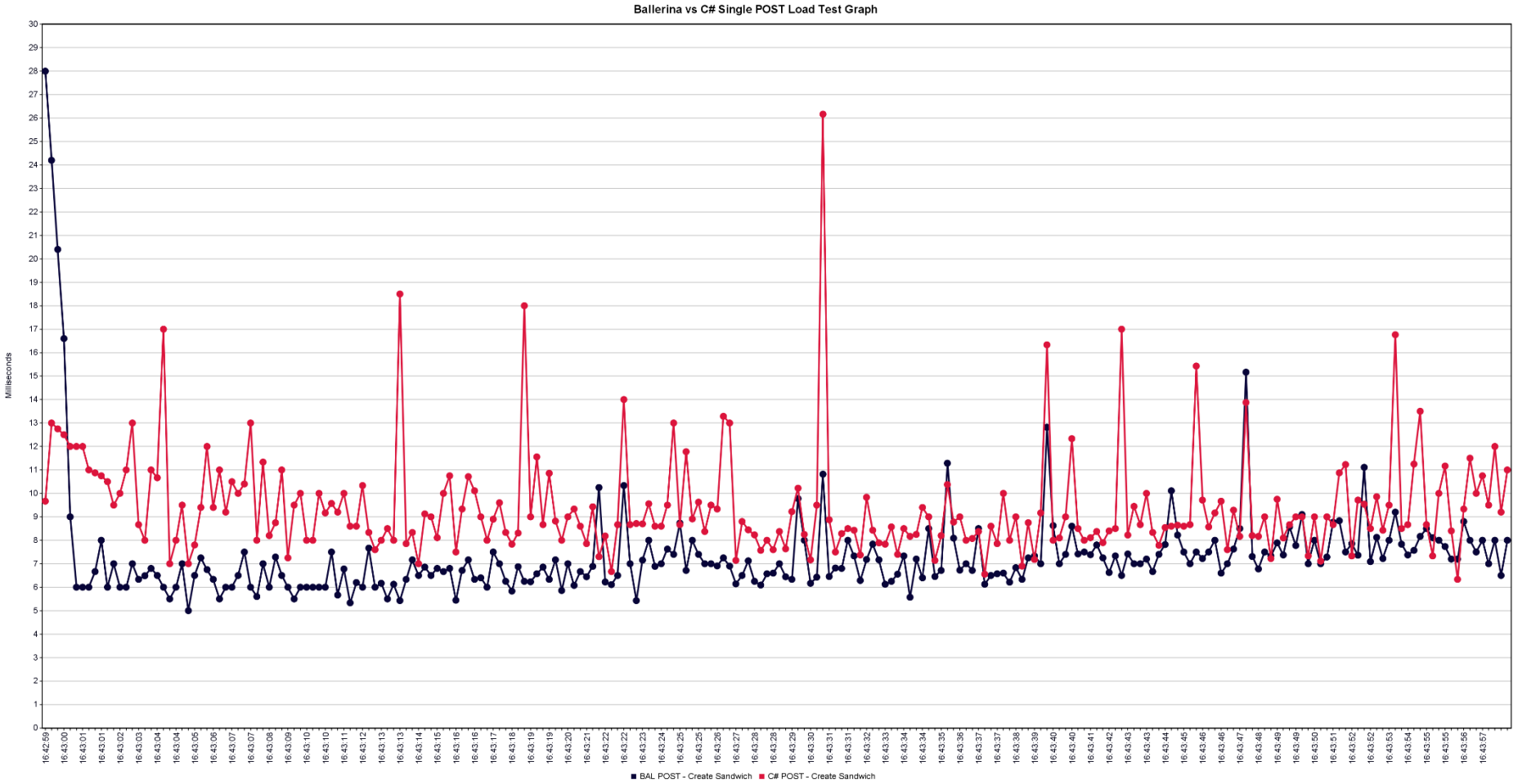


Figure 16 - C# v Ballerina Single POST method response time graph