Microservices Orchestration in Game Development

Orchestrate Multiplayer Game Backend with Ballerina

Marco Nardelotto
Department of Mathematics
University of Padua
Via Trieste, 63, 35121 Padua, Italy
marco.nardelotto@studenti.unipd.it

ABSTRACT

Currently, various software architectures are employed in the video game industry including the use of microservices or the monolithic approach [1]. However, using a monolithic architecture can prevent effective management, maintenance, and scalability, especially as the user base grows. Since the video game industry is expanding every year and requires more and more speed in the development and maintenance of applications in step with the sector and with user expectations, the paper is focused on exploring how to develop a simple multiplayer online game using an architecture based on microservice orchestration to increase the performance of a game system created, in particular to make the entire infrastructure more maintainable, flexible to changes and improve the speed of development and response of the system, using the Ballerina language for helping the orchestration. In particular, the research investigates the use of Ballerina, a Cloud-Native language, for orchestrating backend services in the development of multiplayer video games. Specifically, it focuses on microservices for player matchmaking and game server management. For greater clarity, the article will deal with the implementation choices and analyze the performance and benefits of using Ballerina in the microservices orchestration architecture in two different environments: local and Cloud. Locally, Ballerina will be used both as the main orchestrator and as a support to Kubernetes, thus helping orchestration in an already orchestrated environment, allowing a comparison between declarative orchestration (using Kubernetes) and programmatic orchestration (relying solely on Ballerina). in the cloud environment, the Ballerina language will be used exclusively as the main actor for the orchestration. At the end of the article, the results obtained from this work will be presented.

KEYWORDS

Microservices, Orchestration Architecture, Ballerina, Videogames, Multiplayer, Backend, Cloud.

Abbreviations

PoC: Proof of Concept; TBS: Turn-based strategy.

Acronyms

APK: Android Package Kit; GRPC: Google Remote Procedure Call; HTTP: Hypertext Transfer Protocol; YAML: Yet Another Markup Language.

1 Introduction

Over the past decade the video game industry has grown exponentially and in 2023 the outlook for revenue in the video game industry is US$365.10 billion and is expected to show an annual growth rate (in the years 2023-2027) of 7.24%, resulting in a projected market volume of US$482.90 billion by 2027 [2]. Therefore, being an ever-expanding industry, it is necessary to structure the video games created so that they are easy to maintain, highly available and scalable over time [3]. To do this it is necessary to move from the common idea of creating applications according to a monolithic architecture approach, in which each component of the application is tightly coupled with others, using instead an architecture focused on microservices, which interact with each other. By dividing each component of the video game into microservices, so into smaller and independent components, the following advantages can be obtained:

- Scalability: each service can be scaled independently of the others. This allows us to scale only the services that need more resources, instead of scaling the entire monolithic application. This is particularly important for a multiplayer game, where the number of players can vary greatly, especially in online multiplayer games.
- Resilience: microservices can help the system to be fault-tolerant and resilient to failures, if that microservice doesn't act as a gateway to the other services. If one microservice fails, it does not bring down the entire application.
- Flexibility: microservices can be developed and deployed independently of each other. This means that you can update or add new features to one service without affecting the others. This allows greater flexibility in development and makes it easier to adapt to changing requirements.
- Team productivity: each service can be developed and maintained by a separate team. This allows for greater team productivity and agility, as each team can focus on their own service without being slowed down by the

development of other services, in fact both microservices architecture and multiplayer video games are systems, in which multiple, independent teams come together to work collectively toward a greater goal [4]. Each team can therefore specialize in creating a particular feature of the game, such as a matchmaking or chat system between players.

- Technology diversity: With a microservice architecture, one can use different technologies and programming languages for each service. This allows you to choose the best technology for each service and avoid being tied to a single technology stack.

In the implementation of microservices, it is essential to establish an approach for managing communication among them, along with the chosen architecture. In this case study, I will adopt the microservices orchestration approach, which facilitates communication among individual microservices through an orchestrator entity. Orchestration involves coordinating and controlling the interactions between microservices to achieve a specific outcome. It proves to be particularly interesting as it centralizes the coordination logic, simplifies service composition, and enables efficient scaling and fault tolerance. To manage the interaction between microservices during multiplayer matches in my game, I will leverage the Cloud-native language Ballerina. In this study we will not dwell too much on the use of microservices orchestration in the total creation of the frontend and backend of the game, but we will study how the orchestration of microservices, using Ballerina, can help or facilitate the management of the videogame backend, such as in matchmaking and game server interaction functions. Furthermore, an in-depth comparison will be made to evaluate the different contexts in which Ballerina is employed. This includes both its role as the primary orchestrator in the case of programmatic orchestration, where the logic is explicitly defined so all the code for the orchestration must be developed, and its function as an auxiliary tool for declarative orchestration, where the underlying algorithmic logic is hidden from the user and it is necessary to create a file that describes the configuration for the resource required and then apply the content of the file to Kubernetes.
Specifically, I will implement a PoC that leverages the Ballerina cloud-native language to facilitate the orchestration of microservices in the backend of a simple multiplayer videogame, developed using Unity.
The contributions of this work are as follows:

- An open-source implementation of the implemented orchestration system, including the Ballerina orchestrator, matchmaking server and game server, useful both for educational purposes, and for a hypothetical continuation of development.
- The analysis and testing of the performance of the PoC carried out in the case of a multiplayer video game in 1 v 1 mode.
- Code available in open source in the public domain, at https://github.com/MarcoNarde/microservices-orchestration-with-ballerina-for-videogame-PoC
- The APK and .exe files relating to the application for Android smartphones and Windows PCs placed inside the GitHub repository.

The remainder of this article is organized as follows: Section 2 describes the architecture used for the project and the technological choices for the implementation; Section 3 defines the related work and implementation steps; Section 4 the implemented tests and recorded performances for the work are discussed; Section 5 presents the outcomes of the work, the main problems encountered, and outlines possible improvements that can be made to make the work done more robust; Section 6 draws final conclusions from this work.

2 Architecture and Technical Choices

2.1 Microservices Orchestration Background

This section explains what a microservices orchestration architecture is and why the Ballerina language was chosen to assist the realization of orchestration. Microservice orchestration involves the use of a unit with a special function, called orchestrator or controller, which allows managing synchronously the process flow consistently and sequentially, controlling the execution of microservices tasks in order. This means that each step in the orchestration process runs sequentially, waiting for each previous activity to be completed before moving on to the next. More specifically using orchestration, a composite microservice is introduced to act as a controller, synchronously invoking other microservices using a request/reply protocol to manage the steps of the application process. In an orchestration architecture, the microservices do not embed any knowledge of the application process flow, leaving it solely as the responsibility of the controller [5][6], so orchestration leads to more cohesive solutions, because it is defined and executed on a single system unit instead of being "scattered" across multiple units, but with more coupling and chattiness. The lack of incorporation of process flow knowledge, as defined by this architectural style, is particularly associated with the REST paradigm, which is commonly utilized in distributed systems. Notably, REST does not encompass the concept of a session and operates in a stateless manner. Figure 1 can give a first representation of what architecture is like

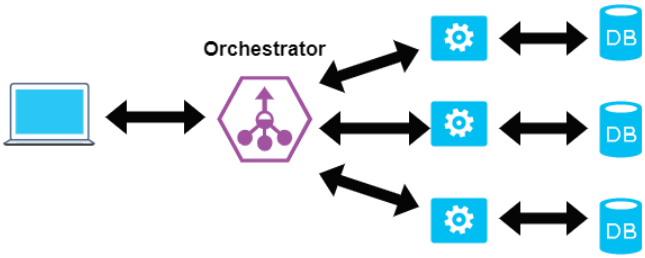

Figure 1 Microservices Orchestration Architecture

The benefits deriving from the use of this type of architecture are [7]:

- Simplicity: orchestration can be simpler to implement and maintain than choreography, in which microservices emit messages upon completion,

triggering one or more microservices [8], as it relies on a central coordinator to manage and coordinate the interactions between the microservices.

- Centralized control: with a central coordinator, it is easier to monitor and manage the interactions between the microservices in an orchestrated system, and to understand where failure events may occur.
- Visibility: orchestration allows for a holistic view of the system, as the central coordinator has visibility into all the required interactions among microservices.
- Ease of troubleshooting: with a central coordinator, it is easier to troubleshoot issues in an orchestrated system.
- Transparency: for the development of microservices in other languages and without seeing the coupling.
- Reusability and maintainability: since microservices are designed to be independent and encapsulate specific functionalities, they can be reused across multiple applications, reducing duplication of efforts, and improving maintainability.
- Enhanced performance and optimization: microservices orchestration allows for fine-grained optimization of individual services. Performance optimizations, such as caching, load balancing, and request routing, can be applied to specific microservices based on their unique requirements, resulting in improved overall system performance.

It's important to note that while microservices orchestration offers these advantages, it also introduces complexity such as scalability, development intricacies, testing challenges, potential bottlenecks, and network-related issues, as well as server registry complexity [9]. Therefore, effective design, seamless implementation, and proficient management are crucial for maximizing the advantages offered by microservices orchestration architecture.

2.2 Project Architecture

The architecture defined for the project is essentially based on the orchestration of 2 other microservices useful for the backend of the video game to be created. Specifically, the two microservices that will be called by the orchestrator will be the matchmaking service, which has the purpose of "pairing" two players to create a lobby, and the game server management service, which allows you to keep track of the games and to independently manage every single game. The orchestrator, implemented using Ballerina, acts as a central system for coordinating operations between clients and their respective services to ensure transparency between the user and the game being played and allows the two services to be invoked to perform more complicated functions without the need for more client requests.

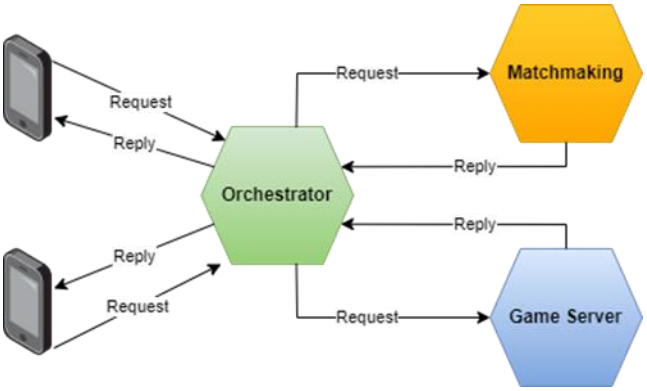

Figure 2 Project Architecture View

Figure 2 shows exactly the base architecture designed for the entire cloud application created. The orchestrator acts as an interface between the different services and clients, receiving client requests and satisfying the different types of requests for each one of them. To interact with the orchestrator, each client will be required to have the game application, implemented using Unity, installed on their device. This architecture has been implemented in the different phases of the PoC, also adapting it to the different environments, in which it has been deployed and tested.

2.3 Why Ballerina

Ballerina is an open-source programming language for native cloud programming, and it was chosen over other cloud-native languages, such as Jolie, for the development of the microservices orchestrator because it facilitates the integration of services in the cloud as well as offering the following features [10]:

- Service composition: it provides a natural syntax for composing services, which means that you can easily build complex applications by combining multiple microservices together. This makes it easy to create a single cohesive application out of many smaller, independent services.
- Client objects allow Ballerina developers to communicate with a remote process that follows a given protocol. The remote methods of the client object correspond to distinct network messages defined by the protocol for the role played by the client object.
- It is type safe with built-in support for popular file formats JSON and XML, and seamless conversions between JSON and user-defined types.
- It provides a unique developer experience to move from code to cloud. The Ballerina compiler can be extended to read the source code and generate artifacts to deploy your code into different clouds. These artifacts can be Dockerfiles, Docker images, Kubernetes YAML files, or serverless functions [11].
- Service discovery and load balancing [12]: Ballerina provides built-in support for service discovery and load balancing, which means that you can easily manage and route traffic to different microservices. This feature

helps to improve the overall performance and reliability of a cloud app.

- It allows visualizing a program written in Ballerina as a sequence diagram. The diagram will display the logic and network interaction of a function or a service resource.

Analyzing the characteristics of the language and the features offered, Ballerina seemed the right language for this PoC, as well as having a more intuitive language syntax, in my opinion, compared to other cloud-native programming languages.

2.4 Why Unity

For the creation of the frontend of the video game and the interaction between it and the orchestrator, I decided to use the Unity game engine because of its popularity and widespread use in the gaming industry [13][14]. It supports multiplatform development for both 2D and 3D games, as well as non-game interactive simulations. Additionally, Unity offers a free version with certain limitations, making it accessible to a wide range of users. One of the key advantages of Unity is its extensive feature set, which includes various functions for game creation and integration with external services. While it can be used, along with its services offerings for multiplayer games, to build the entire structure of a video game, in my case it was only used to facilitate the interaction between the graphical interface and the microservices orchestrator for the backend.

3 PoC Implementation

This section explains my PoC, which consists in using the Ballerina cloud native language to help the orchestration of microservices used in the backend part of a simple game i.e., tic-tac-toe, which will be explained later. Specifically, an orchestrator microservice was implemented with Ballerina to connect the frontend of the game to the various services of its backend. The work focuses on exploring the orchestration of microservices using a cloud-native language, specifically Ballerina, within the context of multiplayer video game development. The goal is to highlight the advantages and disadvantages of using such an approach and understand its implications, rather than creating a fully functional architecture in its entirety. The main steps for implementation will be described, as well as an explanation of some implementation choices and various deployment and testing steps.

3.1 The Game

The following describes the game and the application created to test microservices orchestration. The video game created is based more on the success of the interaction and communication between the various frontend and backend services rather than on the graphic creation of a pleasant to play video game, for this reason the functions are very limited. The game is Tic Tac Toe, a Turn-based strategy (TBS). It is played by two players in 1 vs 1 mode and consists of a three-by-three grid where each player alternately places the marks X and O in one of the nine spaces in the grid, the first player who manages to fill a row, column or diagonal with their symbol wins the match, otherwise ends in a draw. Initially each player will send a request containing their username (and their IP address) to find an opponent with whom to be paired and subsequently on the game screen with each touch in the boxes where to place the symbols, messages will be sent to notify the inserting these symbols into the grid. The game ends when there are no more moves available or when one of the two players wins. In any case at the end of the game each player returns to the main screen. The application acts as a frontend to use the orchestrator services, it is built with the Unity game engine and is designed to be usable by Android smartphones and Windows PCs, as it has not been tested on other platforms. Figure 3 and Figure 4 show two screenshots of the game implemented in PC mode (equivalent to mobile mode)

Figure 3 Initial Game Screen

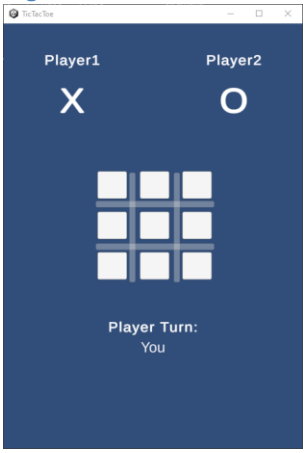

Figure 4 In Game Screen

The first screen in Figure 3 allows the user to enter their username, used to identify the player in the game, and then clicking the "PLAY" button will allow the client to start searching for an opponent and consequently for a game. Once the match has been found, the screen shown in Figure 4 will appear where the two players can start playing. Initially the player who will start will be decided by the server and at each turn, the player will be able to select one of the 9 boxes on the game board to place their own symbol. On the screen, under the grid, there is a label, which defines for each player, which is allowed to move in the current

round, so if the player sees "YOU" in the label, it indicates their turn to move. If they see "OPPONENT" instead, they must wait. Once one of the two players manages to win, or the available moves are exhausted, the game ends showing the result and you return to the first screen.

3.2 System Implementation

As described in previous section, the architecture created is based on the orchestration of microservices mainly linked to the backend of the created application. For this reason, 2 main microservices have been created:

- Matchmaking: a service active on port 8080, implemented using the Go language, which pairs couples of clients wishing to find an opponent. In this case, when two requests arrive from two different clients, the server pairs these clients and returns the pair to the callers.
- Game Server: a service active on port 9090, implemented using the C# language, which defines the game server where the currently active games for the video game are recorded and manages the updates of each game.

The two microservices are then orchestrated by orchestrator microservice developed using Ballerina. In particular, the orchestrator offers two entry points, respectively on ports 9092 and 9095, which allow you to:

- Receive a request to find an opponent and respond with the generated pair.
- Receive a move from a player and update their game and the status of the two contenders.

So, the orchestrator acts as an intermediary between the matchmaking microservices, game servers and clients interfacing using the application in Unity. In this way communications between the microservices will take place in total transparency in the eyes of the application user and will allow me to identify breaking points more easily. The implementations of the two microservices described above will not be discussed, as they do not offer added value to the PoC since the main theme remains the orchestration with Ballerina, in fact, the two microservices have practically the same structure, they listen on a specific port, specifying the paths for the requests and, based on the type of request, they redirect it to the most appropriate method to be processed and return a response.

3.2.1 Orchestrator

In the implementation of the orchestrator, the most significant aspects and characteristics of the Ballerina language will be analyzed and explored to help orchestrate microservices. Specifically, an endeavor was undertaken to construct the orchestrator using Ballerina in such a way that its utilization in the two distinct scenarios (with Kubernetes and as a standalone solution) within the local environment necessitated minimal adjustments and workarounds for adaptation. First, record types were defined within the orchestrator, which are used to define messages in communications between client-orchestrator and microservices-orchestrator.

```
// Player definition
type Player record {|
    string id;
    string username;
|};

// Lobby definition for 2 players
type Lobby record{|
    Player opponent;
    Player you;
|};

// Match definition with match id
type Match record{|
    Player playerX;
    Player playerO;
    Player starter;
    string matchId;
|};

// Move definition for a play of the game
type Move record{|
    Player p;
    int pos;
    string gameKey;
|};
```

Figure 5 Message Records Type

In Figure 5 shows the main types of records used for the exchange of messages, in particular the Player type has been defined to identify the single player who connects to the service, the Lobby type is useful for defining the two players of a game, used as an intermediate record from the orchestrator to create the real game, the Match record, which allows obtaining the data of a game instantiated in the game server, this record contains the definition of the id of the match to be able to identify it, and finally the Move type identify a move a player makes in a game in a specific position. After the definition of the records for the messages, the two services offered were defined, i.e., the service to find an opponent and get the game data and the service to send and receive game data to the game server.

```
service /orchestrator/find on findGameEP {
    Visualize
    isolated resource function get getmatch/[string username](http:Caller caller) returns error? {
        //...

        // Build player information
        Player player = {id: caller.localAddress.ip, username: username};
        // Call the FindOpponent function to get the opponent
        Player opponent = check FindOpponent(caller, player);
        // Call the GetStarter function to start the game server and decide which player starts
        var matchInfo = check GetMatchInfo(player, opponent);
        // Build the response
        var response = new http:Response();
        response.setPayload(matchInfo.toJsonString());
        check caller->respond(response);

        //...
    }
}
```

Figure 6 Find Match Service

The first service shown in Figure 6 allows the program to listen on port 9092 in the *orchestrator/find* path and to receive the request to find an opponent and consequently a game, from a client using the *getmatch* method. Once a request is received, containing the user's username, the service creates the player object to contain

the requester's username and IP address, to be processed by the matchmaking server using the *FindOpponen*t (Figure 7) function and eventually the player object containing the opponent's information will be returned.

```
isolated function FindOpponent(http:Caller caller, Player player) returns Player|error {
    // Create WebSocket connection with matchmaking server
    websocket:Client matchmakingClient = check new ("ws://matchmaking-service:8080/matchmaking",{
        handShakeTimeout: 350
        //...
    });
    // Send to server
    check matchmakingClient->writeMessage(player.toJson());

    // Read message streaming
    string playerResponse = "";
    while (true) {
        string|error message = matchmakingClient->readMessage();
        if (message is websocket:ConnectionClosureError) {
            // Websocker closed connection
            break;
        }else if(message is error){
            // WebScoker send an error
            break;
        }else{
            // Receive opponent from server
            playerResponse = message;
            break;
        }
    }
    string rawData = playerResponse;
    // Get the `json` value from the string.
    json j = check rawData.fromJsonString();
    // Get player data from json
    Player opponent = check j.cloneWithType();
    return opponent;
}
```

Figure 7 Find Opponent Function

The function allows calling the matchmaking service by specifying, in the body of the associated service, also some connection features such as the handshake timeout and the configurations for resending the request. At this point the service calls the *GetMatchInfo* (Figure 8) function to communicate to the game server the information of two new players who intend to start a game.

```
isolated function GetMatchInfo(Player player, Player opponent) returns Match|error {
    // Create http connection with game server's microservice with a configured client
    http:LoadBalanceClient findstarterClient = check new ({{"http://game-server-service:9090",
        timeout: 200
        //...
    }});
    // Build lobby information
    Lobby lobby = {
        opponent: opponent,
        you: player
    };
    // Send request to server
    http:Request request = new;
    request.method = http:POST;
    request.setPayload(lobby.toJson());
    http:Response playerStart = checkpanic findstarterClient->post("/findstarter", request);
    json resp = check playerStart.getJsonPayload();
    Match gameConf = check resp.cloneWithType();
    return gameConf;
}
```

Figure 8 Get Match Info Function

In this case, the function specifies the game server service to connect to. It utilizes a LoadBalanceClient that enables the specification of URLs for redirecting and balancing requests among the replicas of the game server service. Additionally, it allows for the customization of settings such as accepted error codes for the response and actions to be taken in case of a connection failure. This information is processed by the game server, which returns an object of type Match containing the two players, the starter between the two and an id for the game. At this point the service obtains this object and sends it to the two clients waiting for the game. For the second service, listening on port 9095, which allows two clients in a game to send their moves and receive updates on the state of the game, a WebSocket connection is used to maintain a continuous connection between the client and the server, eliminating the need to reestablish the connection with every move.

```
map<map<websocket:Caller>> matchesMap = {};
Run | Debug | Try it | Visualize
service /orchestrator/game on new websocket:Listener(9095) {
    Visualize
    resource function get .(http:Request req) returns websocket:Service|websocket:UpgradeError
            return new WsGameManage();
    }
}

service class WsGameManage {
    *websocket:Service;
    //...
    Visualize
    remote function onTextMessage(websocket:Caller caller, string text) returns error? {
        json j = check text.fromJsonString();
        InitialMessage | error im = j.cloneWithType();

        //...
        //Check for a move
        Move | error m = j.cloneWithType();
        if(m is Move){
            //Create http
            http:Client makemoveClient = check new ("game-server-service:9090");
            // Le invio al server
            http:Request request = new;
            request.method = http:POST;
            request.setPayload(m.toJson());
            http:Response response = checkpanic makemoveClient->post("/addmove", request);
            string resp = check response.getTextPayload();
            map<websocket:Caller> gameMap = matchesMap[m.gameKey] ?: {};
            //Notify move to clients in a game
            if(resp.equalsIgnoreCaseAscii("change turn")){
                foreach var item in gameMap {
                    if(item.isOpen()){
                        if(item.getConnectionId() == caller.getConnectionId()){
                            check item->writeMessage("change turn");
                        }else{
                            check item->writeMessage(m);
                        }
                    }
                }
            }else{
                //Notify Winner
                json r = check resp.fromJsonString();
                Player | error w = r.cloneWithType();
                if(w is Player){
                    foreach var item in gameMap {
                        if(item.isOpen()){
                            check item->writeMessage("winner-"+w.toString());
                        }
                    }
                }
                //...
            }
        }
    }
    //...
```

Figure 9 Game Manager Service

The service in Figure 9 uses a hash-map of game entity, as data structure, in which each game object contains the two clients for that game. For each message that arrives at the service, if it is a message containing a move, it is addressed to the game server in order to obtain the update of the game, the server then replies: with the notification to change the turn, in this way it is sent to the client and the move made is sent, or if the server returns a winner (it can be one of the two players or even null to define a tie), the service notifies the two clients of the winner and will subsequently close the connections with the clients.

3.3 Implementation Steps

3.3.1 Stand-Alone Orchestration

In the first phase of implementation, the entire system was distributed locally on the PC using Docker containers to start the images of the 3 microservices. The entities, visible in Figure 10 and Figure 11, which participate in the whole system, and which are present in the diagrams are:

- The clients: represented by smartphones, which are the devices on which the application is installed and from

which the game can be interfaced, and consequently the services offered.

- The orchestrator: used precisely to make communication between clients and the various servers transparent and to orchestrate the use of their services. In this case it is the only central coordinator of the entire system.
- The two specific servers for matchmaking and game management.

The following images define the two situations, in which the orchestrator and clients can find themselves using the application.

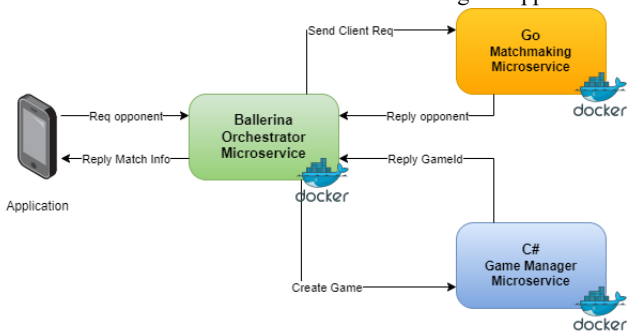

Figure 10 Find Match Phase

Figure 10 shows the system architecture, tested locally, in the finding opponent and game phase. Each client enters the application and through the screen in Figure 3 enters their username and sends a message to the orchestrator to find an opponent, the orchestrator is listening on their services and whenever a player's request arrives:

1. Encapsulates the player's request into a new message to be sent to the matchmaking server and waits for a response from it, i.e., until two clients are matched.
2. It receives the response in the form of a tuple containing the information of the two players and sends it to the game server to instantiate the game and obtain its details.
3. Receives the response from the game server containing the information of the game, i.e., the two players, the game ID to identify communications to it and the player who will start first. That response is then sent to the two clients so that they are in sync with the information.

Finally, the clients get the match information from the orchestrators and manage their interface by switching to the match screen.

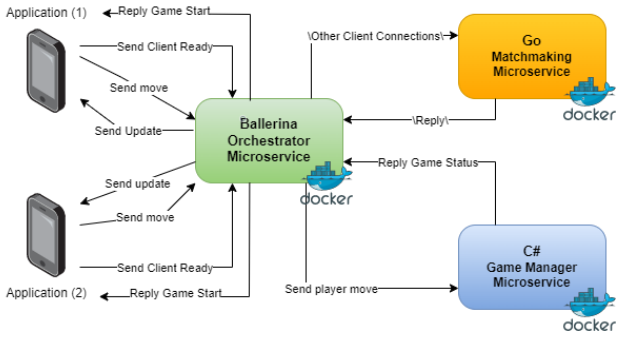

Figure 11 Manage Game Phase

Figure 11 shows the second system state. In this scenario each client is on the game screen (Figure 4), and initially sends the orchestrator, transparently to the player (client), a message to the orchestrator to notify that the client is *"Ready"*. When the orchestrator gets messages from both clients of a particular game, it notifies the clients that they can start playing, establishing a continuous WebSocket connection with such clients. The clients then start sending messages containing the moves made to the orchestrator (based on their turn). The orchestrator receives these messages and redirects the moves to the game server, which will take care of updating the state of the game; the orchestrator waits for a response from the server, which can be:

- A status update notification, in this case the orchestrator notifies the client that sent the move of the request to change the turn, while the other client sends the executed move and defines the new turn.
- A message containing the name of the winner (or tie if the game ends) and this message is forwarded to the two clients to end the game.

When the game ends, the clients disconnect from it and the connections are closed automatically and the game is deleted from the servers. Meanwhile, the orchestrator still satisfies other requests for the creation of new games or player matching. In this first phase of development, the basic concepts of the Ballerina language and the calling of external services in it were explored, as well as methods for coordinating these services and to build docker files and images thanks to the Ballerina's feature called code to cloud. This phase was not only instrumental in establishing the foundation for analysis, but also presented a valuable opportunity to gain a general understanding of the complexities involved in implementing a programmatic orchestration. In fact, every aspect of the orchestration was handled and programmed using Ballerina exclusively.

3.3.2 Cloud Deployment

This paragraph defines the architecture created to move the orchestration system from local to a cloud environment. Specifically, Figure 12 shows the architecture used when the entire local system was moved to the cloud. Requests from all clients are directed to a Linux virtual machine with a public static IP, which has the 3 microservices described above running inside it, i.e., the orchestrator, the game server, and the matchmaking server. In this way the microservices operate as in the previous on-premises architecture but can be accessed using the IP address

of the VM and the reference port of the orchestrator (9092) to obtain requests from the cloud.

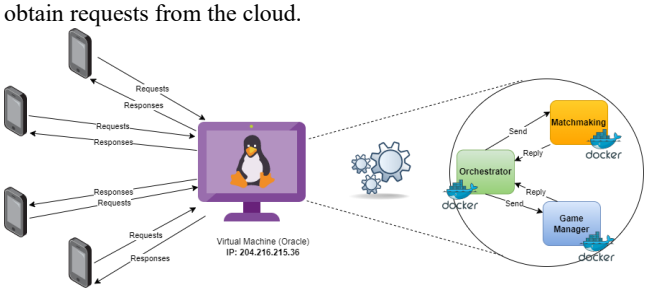

Figure 12 Cloud Architecture

While deploying to the cloud for testing outside of the local environment, I encountered various challenges such as:

1. Finding a free provider for releasing the containers in the cloud.
2. Releasing containers to the cloud and choosing between using a virtual machine or a single service for each container.
3. Establishing communication among the containers.
4. Setting up a cloud environment with a provider.
5. Making two ports of the orchestration service publicly accessible to the outside world.

To solve these problems, I made the choice to use a virtual machine that embeds the microservices containers, rather than using a separate service for each individual container. This decision was made because the virtual machine option was available for free at the time, as opposed to deploying each container in its own environment. A VM from the Oracle provider was chosen, specifically the VM.Standard.E2.1.Micro version, available for free with some limitations on usable memory. This choice was preferred over other providers, like Google, as their free cloud offerings had limitations, specifically in deploying containers that needed to expose multiple ports. Since the orchestrator required two exposed ports, because it offers two services on two different ports which are 9092 and 9095, this constraint posed a challenge, leading to the selection of the Oracle provider. Once the virtual machine was obtained, it was set up to start docker containers, even with Docker Compose, it was therefore necessary to start the configuration via the SSH. Once Docker was installed, the container images previously created and inserted in my Docker Hub account were downloaded into the virtual machine and the 3 containers were started using a docker-compose file with their start settings. In this way the containers are accessible through the IP address of the virtual machine and their ports exposed to the outside and communicate with each other through the local network created for the containers inside the VM. This phase allowed me to face new challenges like:

- How to integrate Ballerina, and consequently the programmatic orchestration, in a cloud environment.
- Handle requests from devices other than the local computer.
- How to best manage user connections during games.

From this I understood that:

- I can integrate Ballerina by leveraging a feature called "*Code to Cloud*," which facilitates deploying applications more efficiently in the cloud. This feature provides a comprehensive guide on the steps involved in deploying applications, ensuring a streamlined and optimized deployment process.
- Requests from multiple devices and user management had to be addressed using unique identifiers to identify the recipients of the responses more easily.

3.3.3 Local Kubernetes Deploy

In the final phase of developing and testing Ballerina's orchestration capabilities, I encapsulated the entire orchestration system with the Ballerina orchestrator, within an already orchestrated system using Kubernetes. This means that the orchestration logic is fully declarative, facilitated by the Kubernetes API. The implemented Ballerina microservice plays a crucial role in supporting the declarative orchestration by managing the process flow. However, it does not handle functionalities such as microservice replication.

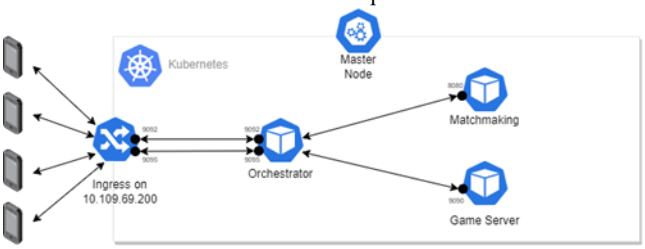

Figure 13 Deploy with Kubernetes

Figure 13 shows the architecture of the system, deployed within Kubernetes. Specifically, 3 pods were created within a cluster managed by a master node, in each of the pods one of the containers for the microservices was inserted and started, then the orchestrator, the matchmaking server and the game server. The pods communicate with each other through services exposed within the cluster and therefore accessible only by other pods within it, while for communications with the outside an input service has been defined (*ingress*), which allows exposing an IP address to access and access ports, in this case the ports of the orchestrator have been exposed in order to make it accessible from the outside and to make communication with other microservices more safety. In particular, the services to which the pods can connect can be found using the appropriate configuration files, which in this case have not been implemented as the system created is quite simple. Overall, the whole system works in the same way previously described in the other paragraphs. Overall, the whole system works in the same way previously described in the other paragraphs, with the orchestrator, which continues to manage the communication sequence with the microservices, however being helped by Kubernetes, which guarantees that at least 1 instance of each service is active to be call. This phase was useful to understand how Ballerina could behave in an declarative orchestrated environment and allowed me to better understand the functioning of some of its features, such as the load balancing system, used when multiple instances of a microservice have been created, and other methods to manage disconnections, connection

timeouts, and failed request resending attempts. This approach of combining Ballerina with Kubernetes appeared highly suitable and advantageous. It enables Ballerina to fully leverage its previously defined features while delegating certain tasks, such as replica management and pod service declaration, to Kubernetes. Ballerina effortlessly utilizes these services to invoke the required microservices without the necessity of specifying precise URLs for each individual service.

4. Performance Evaluation

This section evaluates some characteristics and performances of the implemented system. Will be evaluated:

- The ability of each individual unit of the orchestrator to perform its task without errors occurring in its implementation. Unit tests were therefore carried out to verify the correctness of the implemented code units. This is because I want to automatically verify the correctness of the implemented code through a series of tests. Integrity tests were also conducted across all components of the system in addition to unit tests. However, they have not been included in the paper since their only purpose was to verify the overall reliability of the system, and the tests described in this section are considered sufficiently comprehensive for the PoC.
- The performance of the orchestrator in two distinct scenarios: first, with the orchestrator deployed in the Oracle virtual machine cloud environment, and second, with the orchestrator deployed within a Kubernetes pod in the local environment. Specifically, the evaluation focuses on several performance metrics. These included measuring the response time of the orchestrator to the requests sent, calculating the failure ratio (which represents the ratio of unsuccessful or unresponsive requests to the total number of requests sent), and assessing Ballerina's capability to evenly distribute requests between two microservices offering the same service, thereby determining the percentage of distributed requests on each microservices using a load balancing feature of Ballerina.

These assessments focus on analyzing multiple performance metrics to obtain insights into system behavior and capabilities, which can be useful in understanding how the system is performing now and as a basis for future improvements. Measuring the response time provides an understanding of the system's latency and its ability to handle incoming requests promptly. Additionally, the failure ratio helps in assessing the system's reliability and its ability to handle a high load without compromising its functionality. Then evaluating the load balancing functionality provides insights into the system's ability to efficiently utilize its resources and ensure fair distribution of workloads. From these evaluations I expect to understand better:

- Whether the orchestrator is implemented correctly from a code point of view and whether it performs the desired functions.
- The ability of the system, and of the orchestrator, to be scalable, within the two different environments, being able to make improvements in the future.
- The main vulnerabilities of the system in relation to the fail rate, therefore its ability to remain consistent even in the presence of any errors and to understand how often they can occur. Trying to analyze the percentage of errors that occur within the different environments, which give an estimate of the degree of consistency maintenance.
- The true potential of Ballerina's load balancing function, trying to understand how it works, how it directs the workload to multiple services of the same type, also managing the resources on which the orchestrator runs, also trying to understand whether the balancing is fair between service replicas.

Through these tests, I anticipate evaluating the system's performance in handling moderate workloads and exploring its potential for scalability in the future with the implementation of modifications and optimization strategies. I expect to better understand the behavior of the system from a point of view of scalability and robustness, also trying to detect the parts where changes can be made. Therefore, these tests are used for the sole purpose of evaluating performance in different environments, but not to precisely evaluate the two different types of orchestration, declarative and programmatic, which require a more general view.

4.1 Unit Tests

Although the entire created system is not so complicated, the use of unit tests allowed me to more carefully verify the correctness of various parts that make up the system. In this case some tests have been implemented focusing on the orchestrator, which is the entity on which the greatest attention has been placed within the PoC. In particular, the tests are aimed at testing the services offered by the orchestrator by trying to imitate the behavior of the external services invoked by it during its operations.

The tests can be found in the *main_test* file of the *tests* package within the Ballerina project and to run these tests you need to run the *bal test* command from the terminal.

The tests in Ballerina can be implemented thanks to the use of the *ballerina/test* module, which allows to better manage the creation of the tests and the mocks of the tested objects or functions.

4.2 Performance measurements

4.2.1 Response Time

In evaluating the performance of the system and of the orchestrator, tests were performed on the response time by the orchestrator. A python script contained in the *loadTestScript.py* file was used, which allows several requests to be made to the orchestrator and to receive responses, calculating the response time of each request. The response time therefore comes from the time between sending the request and receiving the response. The tests were carried out both on the local architecture with Kubernetes and on the cloud system to be able to make

comparisons between the two systems. Using the script, the results were obtained in the tables, in which it is possible to view the number of requests, the average response time, as well as the number of correct answers and the relative percentage.

Table 1 Response Times in Kubernetes-Local Deployment

No. Requests	No. Success	Success %	AVG Resp. Time (ms)
50	49	98.00%	1464.67
70	69	98.57%	1103.46
100	97	97.00%	2044.44
120	117	97.50%	1712.76
150	144	96.00%	2168.30
200	191	95.50%	2545.98
250	243	97.20%	1571.26
300	286	95.33%	1824.00
400	394	98.50%	1257.80
500	486	97.20%	1528.18

Table 1 shows the results obtained from the system running locally with Kubernetes. As we can see, as the number of requests made to the Orchestrator increases, including potential time overlaps between them, the response time significantly increases. This behavior is expected since the execution environment is local, and the available resources are limited. Such times obtained would not be optimal in a cloud environment as the entire system would not be suitable for scaling. The success rate of the requests remains constant with good values.

Table 2 Response Times in Oracle VM Deployment

No. Requests	No. Success	Success %	AVG Resp. Time (ms)
200	200	100	254.51
250	250	100	203.54
300	300	100	200.80
400	400	100	209.30
500	500	100	194.31
1000	1000	100	181.05
2000	2000	100	278.95
5000	4991	99.82	372.14
6000	5990	99.83	346.15

Table 2 instead it shows the results obtained from testing the system in the Cloud environment. In this case, as evidenced by the results, the number of requests made exceeds thousands. This is because during the tests, I observed that the response time and the success rate of the responses consistently provided better values compared to the tests conducted in the local environment with Kubernetes. Consequently, I decided to further tests the Cloud environment to determine the threshold at which the system continued to deliver a 100% success rate. It is evident from the results that the cloud environment currently offers better performance compared to the local environment with Kubernetes and success rate is also better than that recorded in the Kubernetes environment. This does not mean that the system is perfectly scalable in a cloud production environment. On the contrary, it serves as a decent starting point for conducting more intensive tests if one truly wants to assess the system's scalability in a real-world situation.

4.2.2 Fail Rate

This paragraph defines the results of the tests carried out on the system both in the cloud and locally with Kubernetes. In particular, the tests are reported to detect the percentage of errors obtained based on the number of requests made on a client, where each test was performed to make N requests done with a python script (*failTestScript.py*) that initiates 10 different threads to send such requests which may be subject to time overlap, for M times to obtain a value, which is the average percentage of errors obtained in the M repetitions. To maintain simplicity and ensure consistency across all sequences of requests, I conducted each test with a fixed number of repetitions, denoted as M = 5. This value was chosen as it struck a balance between being too low, which would render the results insignificant, and being too high, which would lead to longer waiting times during extensive testing. By using this standard value, I aimed to obtain an approximate estimation of the fail rate for the system.

Table 3 Fail Rate in Oracle VM Deployment

No. Requests	AVG Fail Rate
250	0.08%
300	0.13%
400	0.30%
500	0.13%
1000	0.10%
2000	0.13%
5000	0.22%
6000	0.30%

As depicted in Table 3, the percentage of failed requests during repetitions was consistently very low, always below 0.5%. This indicates that the cloud-based system is less vulnerable and more reliable when processing requests, exhibiting a lower rate of errors. The non-linear pattern observed in the failure rate can be attributed to various factors. These factors may include the system's capacity to handle a certain number of requests concurrently, potential bottlenecks, especially in the gateway of the VM, or limitations in resource allocation, or the presence of external factors such as network latency. Further analysis and investigation are necessary to fully understand the underlying causes of the non-linear trend in the observed failure rate.

Table 4 Fail Rate in Kubernetes-Local Deployment

No. Requests	AVG Fail Rate
100	0.5%
120	3.33%
150	3.86%
200	2.30%
250	1.60%
300	9.00%
400	7.00%
500	5.32%

Conversely however, as we can see from the results in Table 4, the system deployed in the local environment using Kubernetes delivers significantly poorer performance compared to tests conducted in the cloud, despite handling a smaller number of requests. These tests indicate that, in this scenario, the system is unsuitable for serving a high volume of requests repeatedly, likely due to the extended latency of individual services. Furthermore, it exhibits a higher likelihood of producing errors in responses and demonstrates lower scalability compared to the cloud-based

system. The non-linear trend in the fail rates observed in this scenario can be attributed to several factors. One possible explanation is that the local architecture with Kubernetes may have limitations in terms of computational power, memory, network bandwidth, and storage capacity, as well as having possible disconnect points between the Kubernetes pods. These limitations can hinder the system's ability to efficiently process many requests, leading to higher fail rates. It is important to note that the local system used in these tests was not explicitly optimized or tested for efficiency. It served as a starting point for analysis purposes and may not have been adequately configured to handle the increasing demands of the tests. Conversely, the cloud-based system used in comparison likely benefits from a more robust and scalable infrastructure, allowing it to handle a higher volume of requests with better performance and lower fail rates.

4.2.3 Load Balancing

The load balancing test of the system was performed only in the local system in Kubernetes, as it was easier to replicate the microservices and therefore obtain different URLs for the microservices of a single type. The use of Kubernetes was essential in replicating microservices effectively. Ballerina does not provide direct replication of microservices that have not been implemented using Ballerina, making it more challenging without Kubernetes. Additionally, obtaining the necessary URLs for identification would have required creating additional containers using Docker and placing them within the same Docker network as the Ballerina orchestrator. Therefore, the test consists in sending requests to the orchestrator towards the opponent search and match creation service, so that we can evaluate how it distributes the requests between two URLs relative to two replicas of the matchmaking microservice.

Table 5 Load Balancing Results

No. Requests	Req. First Client	Req. Second Client
150	49%	51%
300	49%	51%
500	50%	50%
1000	51%	49%
1500	50%	50%
2000	50%	50%
5000	50%	50%

As can be seen from the results in Table 5, the orchestrator manages to balance the requests that arrive in such a way as to address about 50% of the requests on a constant basis to each one of the two replicas. These tests were useful to understand the actual functioning of the load balancing functionality offered by Ballerina, but also to verify that the balancing was fair between the microservices replicas. They were also helpful in understanding the integration of this feature with a declarative orchestration environment using Kubernetes.

5 Outcomes

This section describes the problems encountered and the results obtained from the implementation of the PoC, with particular attention to the outcomes obtained from the use of Ballerina in two different orchestration scenarios and environments and from the deepening in the cloud.

The main problems encountered in the implementation of the PoC can be summarized in:

- Ballerina offers several features for cloud programming and deployment; however, the documentation relating to the deployment, especially in Kubernetes, is not very helpful in some situations and above all the examples, related to the language, found on the web are dated and no longer usable.
- As already mentioned, finding hosting for containers in the Cloud is not easy, especially for test containers like in this case, for which hosting is required to be free, and it is also necessary to choose the type of hosting that you consider most suitable, between hosting the single container or an entire virtual machine hosting them.
- The whole PoC cannot be defined as totally complete, as there would be other tests and maybe Ballerina functions to test, especially in the pure Cloud environment where everything is decoupled, however it is a good starting point for further improvements.

In the next subsection the outcomes obtained with Ballerina will be described in more detail.

5.1 Ballerina Features

Here we describe the main features of the Ballerina language that have been found and used to help orchestrate microservices.

First, the ability to call external services with different types of communication protocols (HTTP, WebSocket, ...) is very useful because it allows having constant knowledge of the type of client, to which the requests will be directed. Then Ballerina provides the ability to create *LoadBalanceClients*, which allow you to define different URLs for a single service, so that requests for a specific service are redirected to multiple servers so as not to congest one. In the definition of this client there are other attributes that can be set to define other settings such as the timeout time before closing the connection, the number of attempts to resend a request, together with the time to wait before resending a request, and also an incrementor for the wait interval, you can then define the types of errors that are accepted as a response from the service, to which you make the request. As already mentioned, Ballerina facilitates deploying its services in platforms such as Docker or Kubernetes simply by creating config files so they will automatically create Dockerfiles for Docker or YAML for Kubernetes, this allows you to save considerable time without having to write these files every time. Ballerina offers additional capabilities, which have not been used in the PoC but have only been seen in theory, to orchestrate microservices in the cloud. Some of these features include the ability to define MySQL-type clients, thus interacting with remote databases; GRPC can be called or defined to make remote calls to functions. It also offers the possibility of interfacing with other types of services such as clients that use FTP, Kafka, RabbitQ or other services.

5.1.1 Ballerina in Declarative & Programmatic Orchestration

This section will explore the advantages and disadvantages of using Ballerina in the two distinct types of orchestration: declarative and programmatic orchestration. Each type will be discussed separately. When using Ballerina as the primary orchestrator in a programmatic orchestration, the following key advantages have been found:

- Fine-grained control over microservice orchestration allows for specific management of aspects such as service invocation via URLs, connection management, and streamlined process flow across the entire system.
- Simplified creation of files for exporting Ballerina code to Docker (and others) containers.
- Enhanced visibility of the entire system through automatically generated diagrams by Ballerina in the compiler, facilitating communication with external services.

Alongside the advantages, there are also certain disadvantages associated with this type of orchestration:

- Replicating non-Ballerina microservices can be challenging.
- Obtaining the URLs of microservices dynamically, especially when they are replaced by replicas in real time, can pose difficulties.
- Maintaining the overall health of the system becomes challenging as there is no inherent mechanism to start or create replicas of a downed service.

Conversely, when Ballerina is combined with Kubernetes in a declarative orchestration, several issues are resolved. In this scenario, Kubernetes enables the management of microservice replicas, ensuring their constant availability and facilitating their replacement in case of malfunctions, improving the scalability of the system. It also allows the assignment of identifiers known as *services* to microservice ports, eliminating the need for volatile URLs. However, using Ballerina with Kubernetes does introduce certain disadvantages, including increased system complexity and the requirement to create configuration files for each microservice intended for deployment in Kubernetes. While Ballerina simplifies the creation of such files, this can be a disadvantage when working with other programming languages. Lastly, deploying Ballerina microservices with Kubernetes requires a functional Kubernetes cluster and associated infrastructure. This dependency on Kubernetes infrastructure may introduce additional overhead and potential points of failure.

5.2 Improvements

Some improvements to get more outcomes from this project are discussed below. First, databases were not used to store data as it was a small PoC and I did not want to waste too many resources generating databases for each type of microservice, however as a future implementation NoSQL database shared between the various microservices could be used of the same type, to make the information always up-to-date and congruent on each node. I could then deploy each microservice to a different virtual machine in the cloud, instead of the same one, this would help you better understand how to find and connect to microservices deployed in the cloud, the next step would be not to use a virtual machine that is expensive in terms of resources, but use a platform like Google Cloud that allows direct deployment of microservices containers in their cloud. Then it could be testing the scalability of the whole system more, adding more replicas of each microservice to better see the balance of the requests routed in the respective microservices and to verify how the orchestrator manages to find and use the largest number of distributed microservices, to do so however, it is first necessary to introduce the use of databases as previously described. Further comparisons could then be made between using Ballerina as the main orchestrator and using it alongside Kubernetes. Perhaps in a more complex environment to better outline the advantages and disadvantages of each type of orchestration.

6 Conclusions

In conclusion, the work carried out allows me to state that the Ballerina language can be used to help orchestrate microservices even by adapting to a particular case such as the orchestration of a backend of an online multiplayer video game. I conducted a comparative analysis of Ballerina's application in two specific types of orchestration: declarative with Kubernetes and programmatic as stand-alone. By exploring these approaches, I identified and evaluated the advantages and disadvantages of utilizing Ballerina in each context. Through this analysis, I aimed to share my firsthand experience and insights gained from employing Ballerina in these distinct orchestration paradigms. In particular, the created system seems to work decently both in a Cloud environment orchestrated with Ballerina, inside a virtual machine, and in a local test environment using Kubernetes, to combine Ballerina with a declarative orchestration paradigm. The tests carried out in the various scenarios give an initial representation of the goodness of the work and its results in action and they can be a good starting point for continuing to explore the topic of the PoC. Furthermore, the correctness of the system, has been evaluated through the unit and integration tests carried out especially on the orchestrator but also on the entire system created, focusing on the fact of creating an application that still works at least for a small number of clients that they use. The final work has some flaws, some of which have already been mentioned but wanted as it was not intended to be a finished and usable product on a large scale, and some improvements that can be made, but it is a good starting point for analyzing the usefulness of the Ballerina language and its use in orchestrating microservices related to the video game industry. Having said this, the final work is satisfactory from my point of view, issues related to the Cloud world and the orchestration of very engaging microservices were addressed, such as the distribution of services and their collaboration in an environment external to that of local machines, as well as to have analyzed and started to know a cloud-native language like Ballerina, which has proved to be very interesting above all for its functions offered for programming "*the cloud*". Lastly, to summarize and confirm my contributions:

- My implementation of an orchestrator in Ballerina, along with the other matchmaking microservices in Go, the game server in C# and the game application built in Unity. These resources are open source and available in the project repository: https://github.com/MarcoNarde/microservices-orchestration-with-ballerina-for-videogame-PoC.
- A use case related to the gaming industry has been shown. Where Ballerina language proves its versatility by serving two critical roles. Firstly, it acts as a valuable complement to existing orchestration environments for microservices. Additionally, Ballerina can function as the primary orchestrator, taking on the responsibility of effectively coordinating and managing the microservices ecosystem.
- The tests and the results obtained from the analysis of the work carried out and the performances observed.

REFERENCES

[1] Njål Nordmark, Software Architecture and the Creative Process in Game Development, Norwegian University of Science and Technology, 2012

[2] https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide

[3] Sofia Estrela, “Microservices Architecture for Gaming Industry Companies”, Instituto Superior Tecnico, Universidade de Lisboa, 2021

[4] https://blog.container-solutions.com/what-video-games-like-apex-legends-can-teach-us-about-teamwork-and-microservices

[5] A. Megargel, C. M. Poskitt and V. Shankararaman, "Microservices Orchestration vs. Choreography: A Decision Framework," 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC), Gold Coast, Australia, 2021, pp. 134-141, doi: 10.1109/EDOC52215.2021.00024.

[6] Chaitanya K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture” International Journal of Advanced Computer Science and Applications(IJACSA), 9(8), 2018. http://dx.doi.org/10.14569/IJACSA.2018.090804.

[7] https://camunda.com/blog/2023/02/orchestration-vs-choreography/#benefits-of-orchestr-rnqu.

[8] Rajasekharaiah, C. (2021). Microservices: What, Why, and How?. In: Cloud-Based Microservices. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-6564-2_2

[9] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. 2017. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In Proceedings of the XP2017 Scientific Workshops (XP '17). Association for Computing Machinery, New York, NY, USA, Article 23, 1–5. https://doi.org/10.1145/3120459.3120483

[10] https://ballerina.io

[11] https://ballerina.io/learn/run-in-the-cloud/code-to-cloud/code-to-cloud-deployment/

[12] https://github.com/ballerina-attic/loadbalancing-failover

[13] Almansoury, F., Kpodjedo, S., & El Boussaidi, G. (2022). Game development topics: A tag-based investigation on game development stack exchange. Applied Sciences, 12(21), 10750.

[14] Hussain, F. (2020). Unity Game Development Engine: A Technical Survey. University of Sindh Journal of Information and Communication Technology , 4(2), 73-81. Retrieved from https://sujo.usindh.edu.pk/index.php/USJICT/article/view/1800.