FANTASY KICKOFF

Final Year Project 2025

Liam Flynn

200098690

Supervisor: Robert O'Connor

BSc (Hons) in Applied Computing

Table of Contents

1. Introduction

1.1 Project Overview

This paper provides an in-depth report on my final year project I developed, a mobile game, “Fantasy Kickoff”. The project was designed to incorporate a wide range of concepts I’ve learned throughout my computer science degree, including game development, UI/UX design, API Integration and cloud-based server architecture using AWS and Node.js, as well as allowing me to explore new topics such as multiplayer integration.

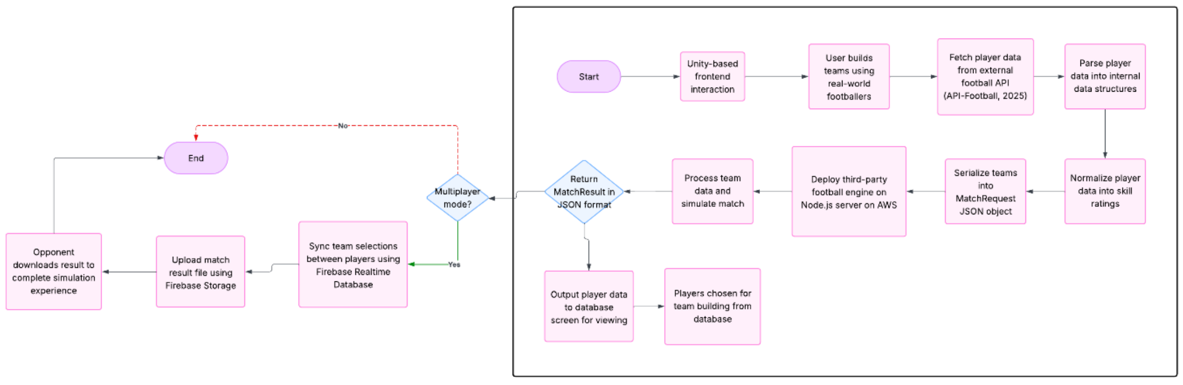
1.2 Project Goals

- Develop a mobile football simulation game built in Unity.
- Display a database of real-world footballers to users.
- Enable users to create custom teams using players from different leagues and teams.
- Implement user authentication to support data tracking and multiplayer features.
- Derive footballer skill ratings from real-life performance data using an external football API.
- Allow players to run custom single player simulations with footballers from different leagues across Europe.
- Allow users to participate in online multiplayer matches against their friends.

1.3 High-Level System Architecture

The system is composed of four main components: a Unity-based frontend, an external football API (API-Football, 2025), a custom Node.js server, and Firebase services for multiplayer support (Firebase, 2025). Users interact with the Unity interface to build teams using real-world footballers and initiate match simulations. Player data is fetched from the football API, and normalized into skill ratings. These teams are serialized into a MatchRequest JSON object.

The match simulation is handled by a third-party open-source football engine sourced from GitHub (footballSimulationEngine, 2022), which I deployed to my own Node.js server hosted on AWS. This engine processes the team data and returns a MatchResult in JSON format. In multiplayer mode, Firebase Realtime Database is used to sync team selections between players, while Firebase Storage enables the host to upload the match result file, which the opponent downloads to complete the simulation experience.


Start
Unity-based frontend interaction
User builds teams using real-world footballers
Fetch player data from external football API (API-Football, 2025)
Parse player data into internal data structures
Normalize player data into skill ratings
Serialize teams into MatchRequest JSON object
Deploy third-party football engine on Node.js server on AWS
Process team data and simulate match
Return MatchResult in JSON format
Output player data to database screen for viewing
Players chosen for team building from database
Multiplayer mode?
No
Yes
Sync team selections between players using Firebase Realtime Database
Upload match result file using Firebase Storage
Opponent downloads result to complete simulation experience
End

2. API and Data Processing

2.1 Football API Overview

The football API I selected for this project was part of the larger API-Sports platform, which provides a range of real-time and historical sports data services. One of the first tasks in my development process was to evaluate various football APIs to determine which would best meet the needs of my simulation system — specifically, one that offered reliable, up-to-date, and detailed player statistics.

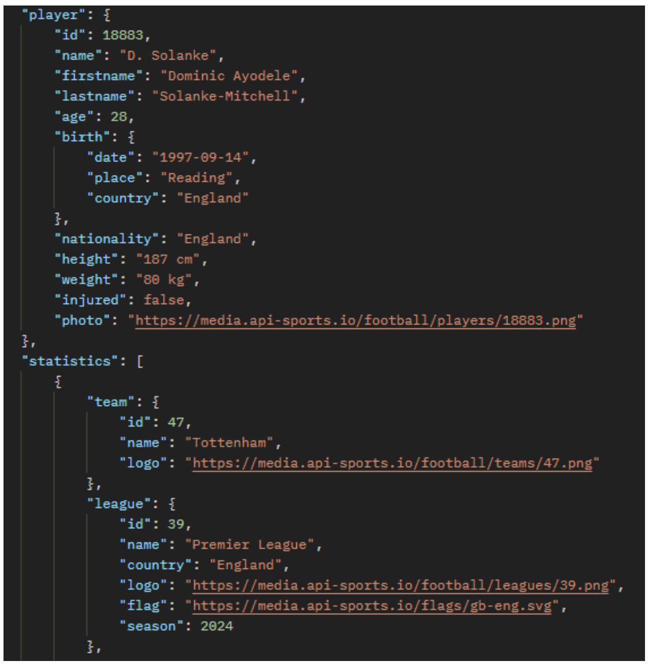
After comparing several options, I chose ‘API-Football’ due to its weekly updates, broad coverage of European leagues, and detailed player data. It supports all major teams and leagues across Europe and offers rich statistical breakdowns for individual players, including offensive, defensive, and fitness metrics.

Another major reason for selecting API-Football was its support for C# bindings, making integration with Unity significantly easier. The API offers multiple endpoints for retrieving data by league, season, team, or individual player. This flexibility allowed me to construct a dynamic and searchable player database for my simulation system.

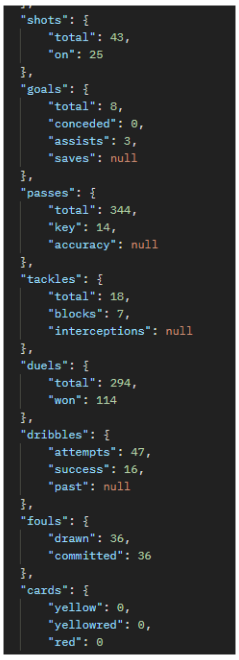
The API documentation is available here:

https://www.api-football.com/documentation-v3

Below is a screenshot of a sample JSON response for a player request, showing the level of detail provided by the API.



```json
"player": {
    "id": 18883,
    "name": "D. Solanke",
    "firstname": "Dominic Ayodele",
    "lastname": "Solanke-Mitchell",
    "age": 28,
    "birth": {
        "date": "1997-09-14",
        "place": "Reading",
        "country": "England"
    },
    "nationality": "England",
    "height": "187 cm",
    "weight": "80 kg",
    "injured": false,
    "photo": "https://media.api-sports.io/football/players/18883.png"
},
"statistics": [
    {
        "team": {
            "id": 47,
            "name": "Tottenham",
            "logo": "https://media.api-sports.io/football/teams/47.png"
        },
        "league": {
            "id": 39,
            "name": "Premier League",
            "country": "England",
            "logo": "https://media.api-sports.io/football/leagues/39.png",
            "flag": "https://media.api-sports.io/flags/gb-eng.svg",
            "season": 2024
        },
```



```json
"shots": {
    "total": 43,
    "on": 25
},
"goals": {
    "total": 8,
    "conceded": 0,
    "assists": 3,
    "saves": null
},
"passes": {
    "total": 344,
    "key": 14,
    "accuracy": null
},
"tackles": {
    "total": 18,
    "blocks": 7,
    "interceptions": null
},
"duels": {
    "total": 294,
    "won": 114
},
"dribbles": {
    "attempts": 47,
    "success": 16,
    "past": null
},
"fouls": {
    "drawn": 36,
    "committed": 36
},
"cards": {
    "yellow": 0,
    "yellowred": 0,
    "red": 0
```

2.2 Fetching and Caching API Data in Unity

To integrate API-Football into my Unity project, I used UnityWebRequest to send asynchronous HTTP GET requests to the API's endpoints. These requests included the required headers, such as the API key, and used query parameters to filter data by league, season, or team. Once the data was retrieved, the JSON responses were parsed into C# data structures like PlayerData and TeamData, using the Newtonsoft.Json library.

Although it was initially a challenge as this was my first time working with UnityWebRequest, it gave me hands-on experience with networking inside a game engine for the first time, which was great to learn and very interesting.

I had worked with APIs during my placement last year, which gave me a solid foundation in handling authentication headers, query parameters, and interpreting JSON responses. While that previous experience was in a web development context, many of the same principles applied here, and it helped me get up and running more confidently in Unity.

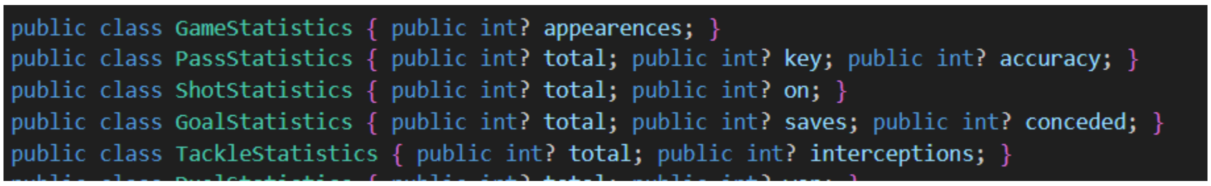
To help understand how the API worked, I used Postman throughout development from start to finish. It allowed me to test different endpoints and study the structure of the JSON responses before trying to implement them in code, as well as send POST requests when needed. This made it much easier to identify exactly what data I needed and how it would be formatted.

At the recommendation of my supervisor, I implemented a local caching system to improve efficiency and reduce reliance on constant API access. After a successful API call, the raw JSON data gets saved to a file inside Unity's persistent data path. On future launches, the application checks for a local cached version before making a new API request. This minimized unnecessary requests and helps to avoid hitting the API's rate limits during development and testing.

2.3 JSON Parsing into Unity Classes

Once I had the raw JSON data from the API, the next step was to convert it into something Unity could work with so changes could be made to the data, ie. changing player's statistics into their skill ratings for the simulation. I used the Newtonsoft.Json library to parse the JSON into C# objects. I created custom data classes like PlayerData and Skill, which matched the structure of the API responses. This made it much easier to work with the data inside the game.

An example of transferring the JSON data into C# objects:



```
public class GameStatistics { public int? appearences; }
public class PassStatistics { public int? total; public int? key; public int? accuracy; }
public class ShotStatistics { public int? total; public int? on; }
public class GoalStatistics { public int? total; public int? saves; public int? conceded; }
public class TackleStatistics { public int? total; public int? interceptions; }
```

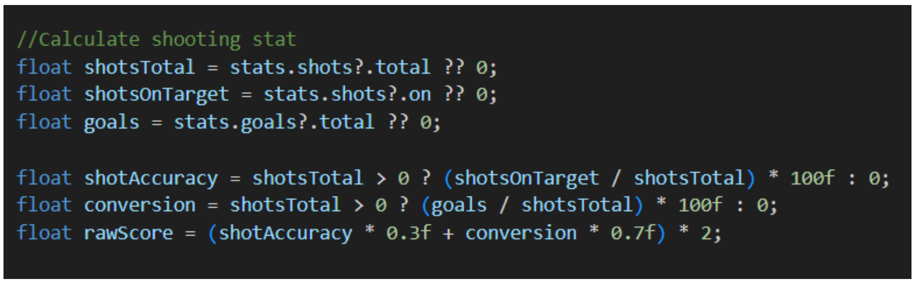
2.4 Normalizing and Converting Statistics to Player Skills

The simulation engine I used expected each player to have eight specific skill ratings: Passing, Shooting, Tackling, Saving, Agility, Strength, Penalty Taking, and Jumping — all on a scale from 1 to 99. To match this format, I had to take raw statistics from the football API and convert them into something the engine could actually work with.

I created my own formulas to map the real-world stats to each of the required skills. For example, the Passing stat was influenced by things like total passes, pass completion rate, and accuracy, while Shooting took into account shot totals, goals, and shot accuracy. For goalkeepers, I used metrics like saves and clean sheets to determine their Saving value.

Once the values were calculated, they were scaled into the 1–99 range to match the engine.

An example of the conversion of statistics into skills:



```
//Calculate shooting stat
float shotsTotal = stats.shots?.total ?? 0;
float shotsOnTarget = stats.shots?.on ?? 0;
float goals = stats.goals?.total ?? 0;

float shotAccuracy = shotsTotal > 0 ? (shotsOnTarget / shotsTotal) * 100f : 0;
float conversion = shotsTotal > 0 ? (goals / shotsTotal) * 100f : 0;
float rawScore = (shotAccuracy * 0.3f + conversion * 0.7f) * 2;
```

2.5 Image Downloading and Caching

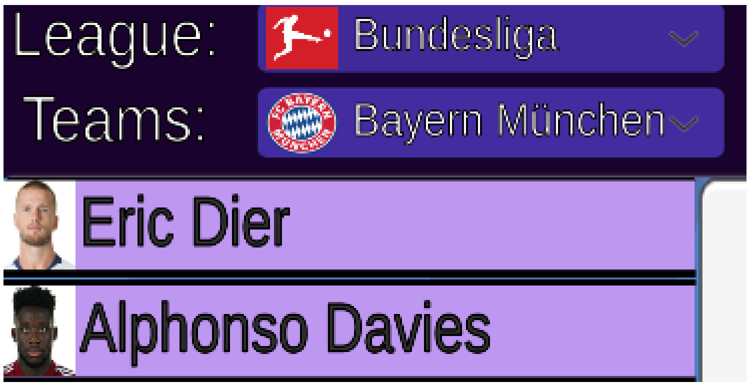
As part of the player database, I included images of the players in each of their individual rows. These images were provided as URLs in the API responses, so I needed a way to download them in Unity and display them within the UI.

Using UnityWebRequestTexture, I downloaded each image from its URL and converted it into a texture that could be displayed on screen. Since many players shared similar names, having profile pictures helped with clarity and user experience.

To avoid re-downloading the same images on every launch, I implemented a basic caching system. Once an image was downloaded, it was saved locally as a PNG file in Unity's persistent data path. On future loads, the application first checks if the image already exists locally before making a network request.

I delete the cache entirely once a week however, to allow for players' new statistics to affect their skills.

An example of the player items derived from the API and images, represented in game within a scroll view:



3. Match Simulation Engine

3.1 Simulation Architecture (Client/Server Split)

The simulation engine used in this project was a third-party, open-source football match simulator sourced from GitHub. Rather than integrating the simulation directly into Unity, I chose to run the engine on a custom Node.js server hosted on AWS. This allowed for a clear separation between the game client and the simulation logic, while also enabling consistent simulation results in both single-player and multiplayer contexts.

Unity acts as the client, responsible for preparing and sending simulation requests in JSON format. These requests contain all necessary player data and team structures. The server receives the request, processes it using the simulation engine, and returns a structured match result in JSON format. This architecture not only made debugging and testing easier but also laid the groundwork for potential scalability, as matches can be simulated independently of the client environment.

In multiplayer matches, only the host sends data to the server. The resulting match output is then shared with the opponent using Firebase Storage, ensuring both players view the same simulation outcome.

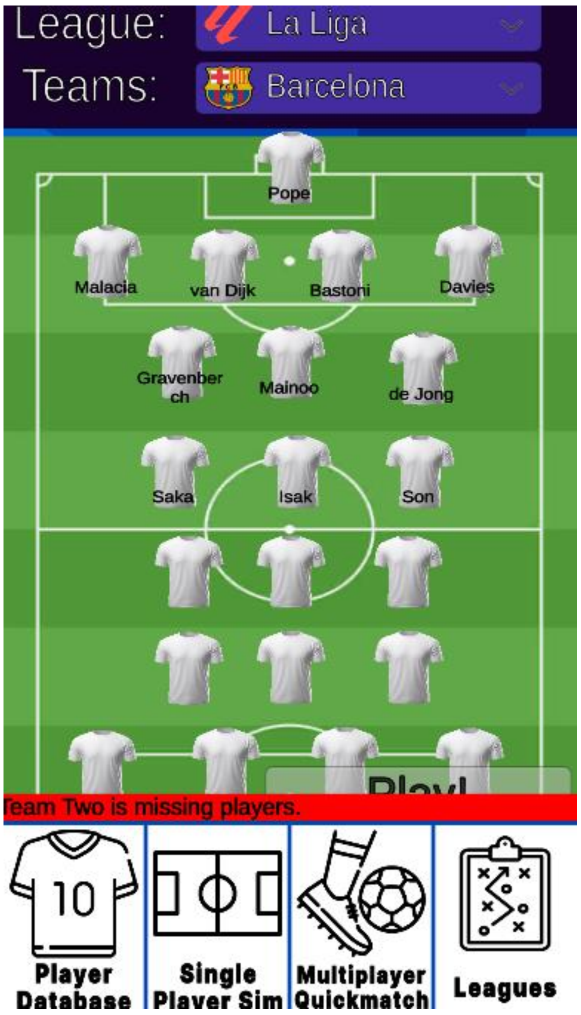
A significant portion of my time was spent on the client-side logic to prepare these requests. This involved allowing the user to manually select and assign real-world players to specific roles in the team using a Unity-based UI system. The PlayerSelectionManager class handled this functionality, assigning each player a position (e.g., ST, CM, CB) and calculating their exact starting Vector2 coordinate on the pitch based on that role. Ensuring the logic was strict and predictable was crucial, as even minor inconsistencies in layout or missing players could cause the engine to crash.

3.2 MatchRequest & MatchResult JSON Format

The communication between the Unity client and the simulation server is handled entirely through structured JSON. When a match is triggered, Unity generates a MatchRequest object that includes two teams, each made up of exactly 11 players, along with pitch information such as pitch width, height, and goal size. These values were all required to match strict expectations set by the simulation engine.

Each player in the request is assigned a position on the pitch in the form of 2D coordinates. These positions had to make logical sense relative to the player’s role (e.g., defenders further back, strikers further up), and had to stay within realistic pitch boundaries. Getting this right was critical, as mismatches in position layout or minor deviations in pitch size often caused the engine to crash without a helpful error message. Through testing and trial-and-error, I found stable values and built Unity-side validation to prevent invalid match setups from being sent.

The single player UI, which allows players to select two teams of completely custom players, and pit them against each other:



Team Two is missing players.

In MatchDataManager, these players were collected and organized into a MatchRequest, serialized with Newtonsoft.Json, and sent to the server. Care had to be taken to ensure the correct structure, especially since the simulation engine required a precise layout: two full teams of 11 players, with correctly formatted skills and coordinates.

The simulation engine processes the MatchRequest and returns a MatchResult JSON object. This initially contained high-level results — goals scored, team outcomes, and individual player statistics. However, early on in development, I realized that having access to detailed match events would be incredibly helpful for both debugging and potential visualization.

To support this, I modified the simulation engine and server to log every player's movement, ball position, and each change that occurred over time. Matches are played out in discrete movements called iterations, with each iteration representing a tick of simulated activity. I updated the server to return not just final results, but also a full breakdown of player positions and ball locations per iteration. This gave me a frame-by-frame view of the match and allowed for deep inspection of what was happening at each step, which would later be used for playback and animation.

3.3 Server-side Execution and Logic Flow

The simulation server was built using Node.js and hosted on an AWS EC2 instance. Its main purpose was to receive a MatchRequest from Unity, process it using the football simulation engine, and return a structured MatchResult JSON. The server ran an Express.js application that exposed a single POST endpoint (/simulate), which accepted the full match data, including both teams and pitch settings.

Upon receiving a request, the server initialized the match and began stepping through the simulation in discrete movements called iterations. Each iteration represented a single unit of simulated time, during which player movements, ball actions, tackles, and goals were processed. The match continued for up to 10,000 iterations, switching to the second half automatically at iteration 5,000.

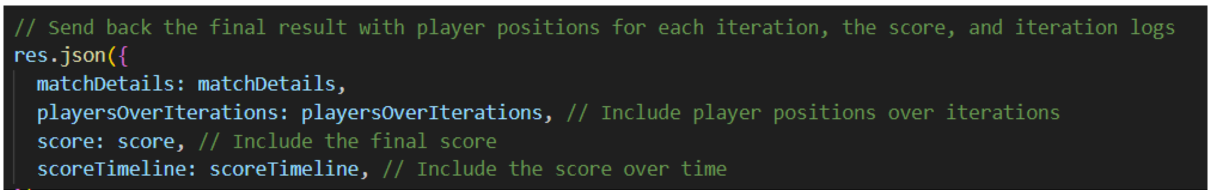
To capture a detailed view of the simulation, I modified both the engine and server logic to log the positions of every player and the ball at each iteration. This data was stored in arrays like playersOverIterations and ballOverIterationsHistory. I also tracked the score over time using a scoreTimeline, recording whenever a goal was scored and at which iteration it occurred.

This detailed logging allowed me to replay the match frame-by-frame in Unity and inspect how player positions and decisions evolved over time. It also made debugging and balancing the simulation significantly easier, as I could see exactly when and why certain events occurred.

The final server response included:

- The full matchDetails object
- Iteration-by-iteration player positions
- A timeline of score changes
- The final match score

This structured output was then parsed on the Unity client and saved locally for display and animation.



```
// Send back the final result with player positions for each iteration, the score, and iteration logs
res.json({
  matchDetails: matchDetails,
  playersOverIterations: playersOverIterations, // Include player positions over iterations
  score: score, // Include the final score
  scoreTimeline: scoreTimeline, // Include the score over time
```

3.4 Response Handling and Local Save

Once the simulation was complete, Unity's MatchDataManager handled the server's response using UnityWebRequest. The response included the final score, detailed player stats, and every frame of the match simulation. I saved this data locally using File.WriteAllText() so that it could be reused offline or replayed at any time.

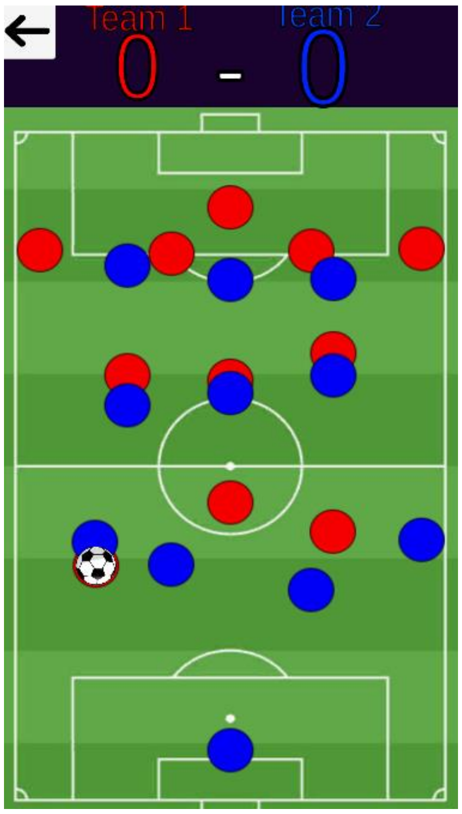
In multiplayer, only the host triggers the simulation. The result is uploaded to Firebase Storage and downloaded by the opponent. This ensures both players work from the same match data, preserving fairness and accuracy.

3.5 Match Playback and Animation

The saved JSON is loaded and visualised in the MatchAnimator class. This script parses the full iteration history and instantiates players and a ball using Unity UI elements. Each player is animated using Lerp between positions frame-by-frame, based on their recorded movement.

Players on both teams are color-coded and placed accurately based on engine coordinates. The animation runs through all iterations at high speed, smoothly transitioning player positions and updating the match score using the stored timeline. I created internal classes like AnimatedPlayer and AnimatedBall to handle their movement logic independently.

A still of the match animation:



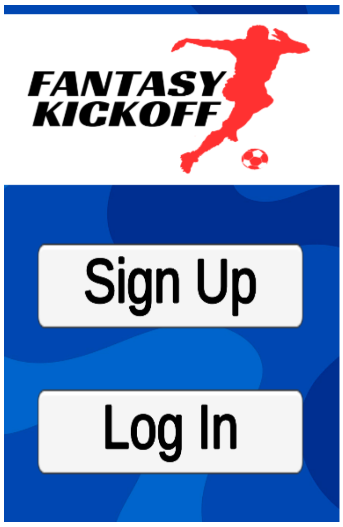
4. Multiplayer System with Firebase

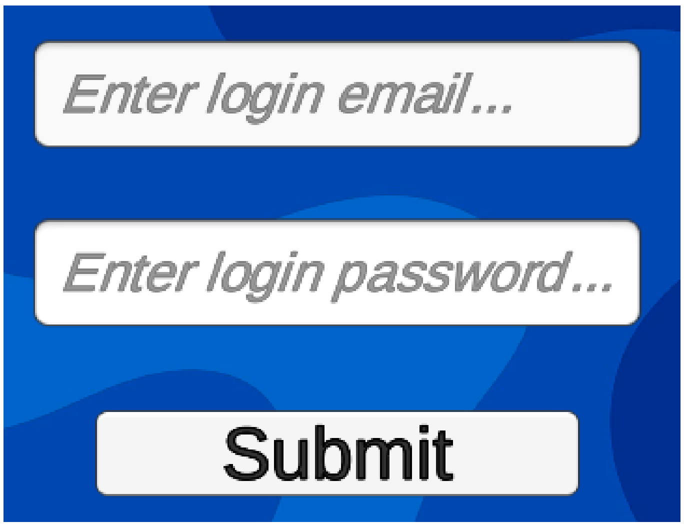
4.1 Firebase Authentication and Lobby System

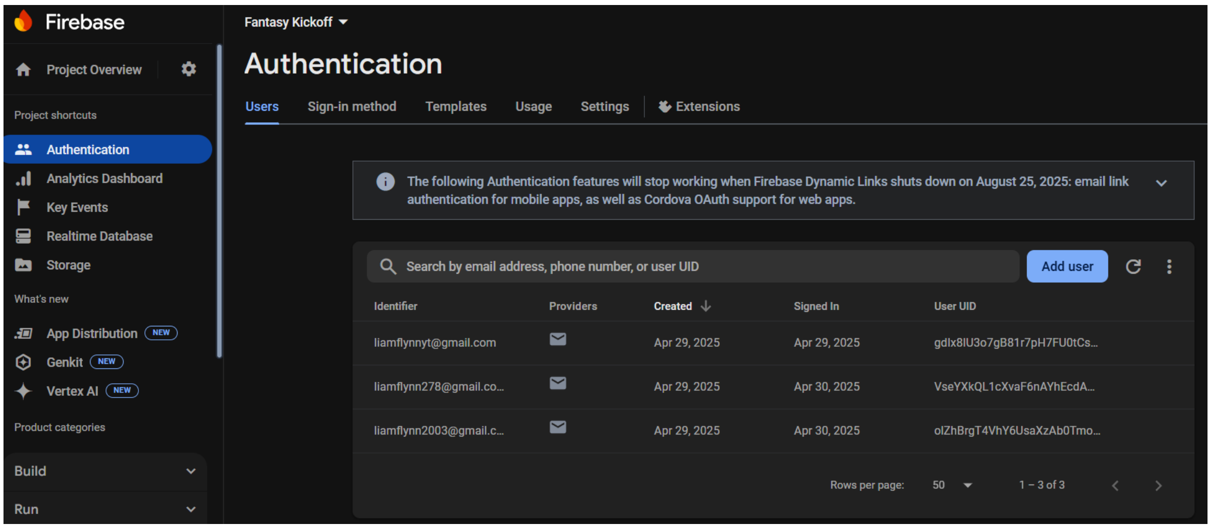
Firebase Authentication was used to manage user sign-in and identity across sessions. Players could register and log in using email and password, which created a unique user ID linked to their actions in multiplayer mode.

After authentication, players could either host a match or join an existing one using a generated lobby ID. The lobby system was managed using Firebase Realtime Database, where each lobby was represented as a node containing information about both players and their selected teams. This allowed two users to synchronize and connect without needing traditional networking or socket-based communication.

The signup/login screen, followed by the login input options:







The following Authentication features will stop working when Firebase Dynamic Links shuts down on August 25, 2025: email link authentication for mobile apps, as well as Cordova OAuth support for web apps.

Search by email address, phone number, or user UID

Identifier	Providers	Created	Signed In	User UID
liamflynnyt@gmail.com	✉	Apr 29, 2025	Apr 29, 2025	gdIx8lU3o7gB81r7pH7FU0tCs...
liamflynn278@gmail.co...	✉	Apr 29, 2025	Apr 30, 2025	VseYXkQL1cXvaF6nAYhEcdA...
liamflynn2003@gmail.c...	✉	Apr 29, 2025	Apr 30, 2025	oIZhBrgT4VhY6UsaXzAb0Tmo...

Rows per page: 50 1 – 3 of 3

4.2 Realtime Team Data Sync (Database)

When each player completed their team selection, the full lineup was converted to JSON and uploaded to Firebase Realtime Database under their respective lobby node. Each team was saved under teamOne or teamTwo, depending on whether the user was the host or the opponent.

I used listeners to monitor the lobby in real time. Once both teamOne and teamTwo existed in the database, the game progressed to simulation. This setup worked reliably and allowed both players to independently build teams before syncing them in the cloud.

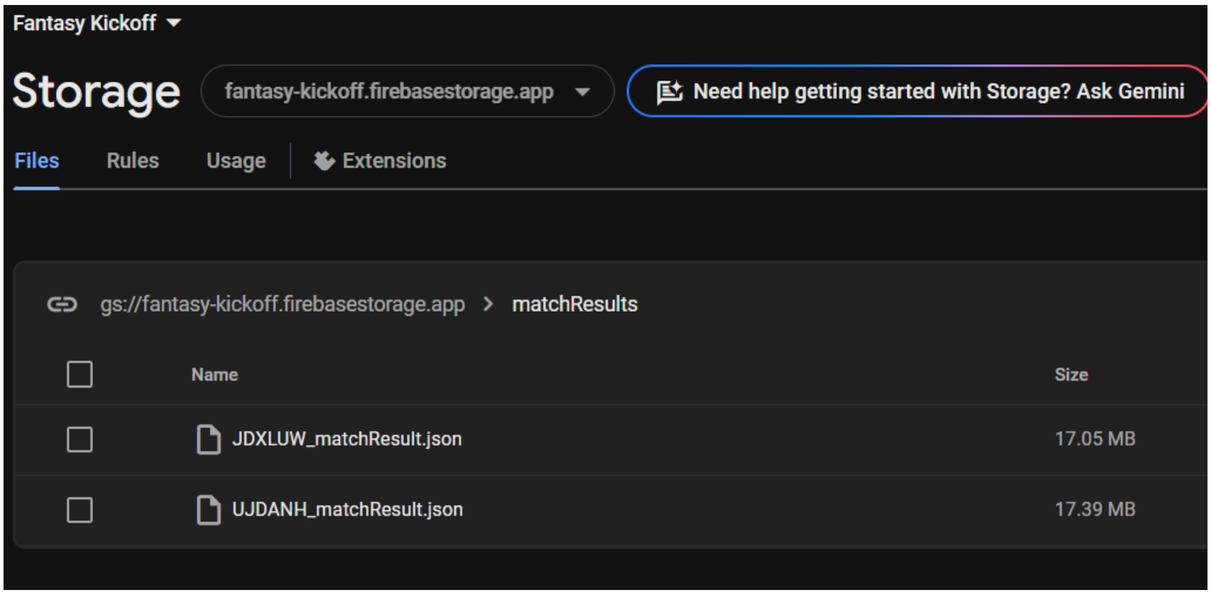
Handling this properly was more challenging than expected. The same team selection UI and logic from single-player mode had to be adapted carefully for multiplayer, making sure the correct players were assigned to the correct positions, serialized accurately, and synced under the right node in the database.

4.3 Simulation Trigger and Firebase Storage Upload

Once both teams were uploaded and confirmed, the host triggered the simulation using the same system as in single-player, sending the request to the Node.js server. After receiving the simulation result, the host player uploaded the entire MatchResult JSON to Firebase Storage.

Originally, I considered using the Realtime Database for this as well, but due to the size of the match data (up to several MB depending on match length and logging), I switched to using Firebase Storage, which is designed for larger files.

The upload process was handled using Unity's Firebase SDK and included converting the JSON to bytes and placing it in a unique path under matchResults/{lobbyId}_matchResult.json.


Fantasy Kickoff
Storage
fantasy-kickoff.firebasestorage.app
Need help getting started with Storage? Ask Gemini
Files
Rules
Usage
Extensions

gs://fantasy-kickoff.firebasestorage.app > matchResults

Name	Size
JDXLUW_matchResult.json	17.05 MB
UJDANH_matchResult.json	17.39 MB

4.4 MatchResult File Retrieval by Opponent

The opponent player continuously checks Firebase Storage for the existence of the match result file. I wrote a coroutine that repeatedly attempted to get a download URL using Firebase's GetDownloadUrlAsync() method. Once the file was available, it was downloaded using UnityWebRequest.Get() and saved locally, just like in single-player mode.

This polling approach was chosen over event-driven methods due to occasional propagation delays in Firebase Storage. By waiting indefinitely instead of relying on a fixed number of retries, I ensured that the opponent would always eventually receive the file, even if there was a delay in upload availability. Both users would view the simulation at the same time, and because of this method, it would be the same match, rather than triggering a separate simulation for each player.

5. Unity Gameplay and UI

5.1 Player Selection UI

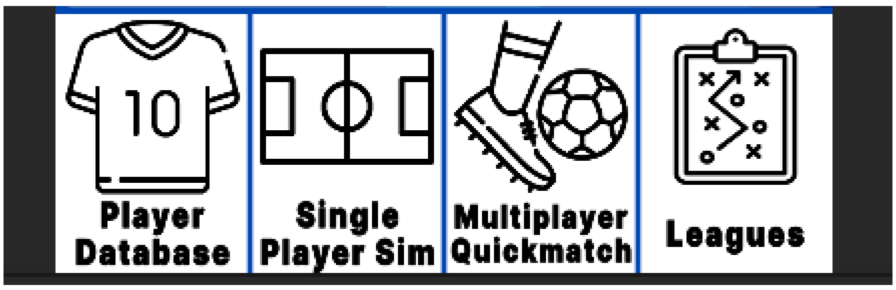
After launching the game, users are greeted with a homescreen displaying the logo and simple options to either sign up or log in. Once authenticated through Firebase, the user is brought to the Main Hub, which serves as the central UI for all in-game features.

From here, a persistent bottom navigation bar allows users to move between three core sections of the game:

- Player Database – A searchable list of all players currently available in the game.
- Single-Player Simulation – A mode where users can create and run their own matchups.
- Multiplayer Quickmatch – A feature that lets players join or create a lobby and simulate a match against a friend.

All of these sections exist within the same scene, and transitions between them are handled using UI toggles, not scene changes. This design choice improved performance and kept the experience smooth and responsive.

In addition to the three main sections, there is also a fourth navigation tab titled Leagues. This was a planned feature intended to allow users to view league-level statistics or simulate mini-tournaments. Due to time constraints, it was not fully implemented for this version of the project and currently displays a "Coming Soon" message. However, I plan to finish this feature in time for the expo in May, and I see it as part of the ongoing development I intend to continue after this project is submitted.



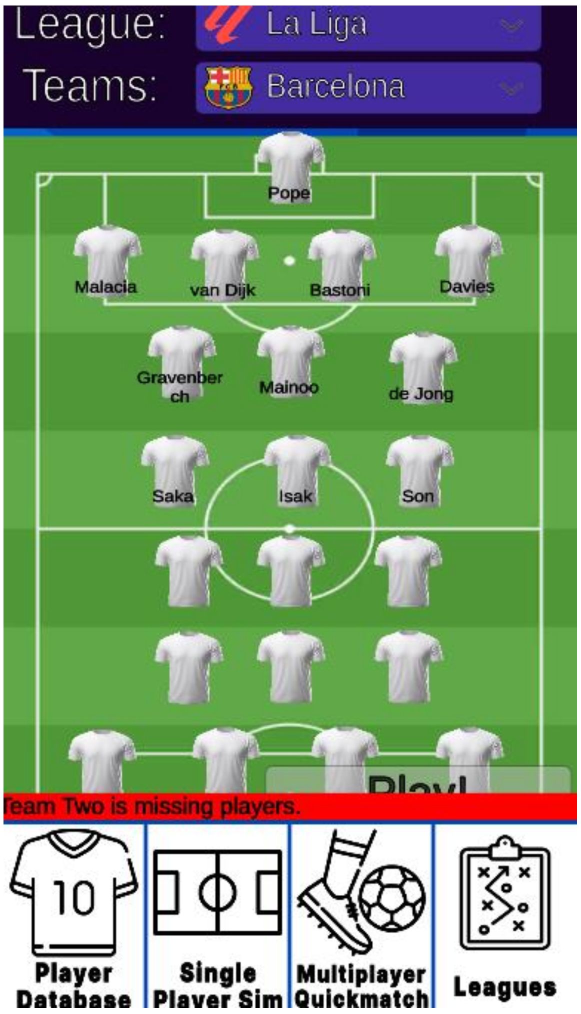
5.2 Player Selection and Team Building

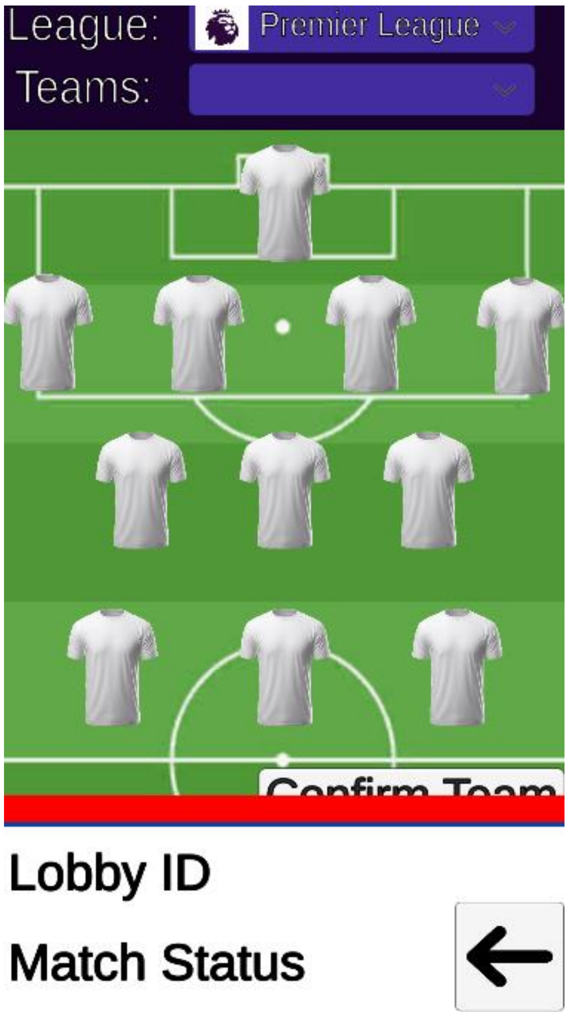
Team selection is a core mechanic shared between both single-player and multiplayer modes. In each case, the user is presented with UI slots corresponding to specific player roles (e.g., GK, CB, ST), and can populate each slot by choosing a footballer from the database.

Behind the scenes, this is handled by the PlayerSelectionManager script, which maps each slot to a logical position label and a physical coordinate on the pitch. These values are essential, as they are directly serialized into the MatchRequest JSON sent to the simulation server.

This system was challenging to implement due to the need to validate player uniqueness, role correctness, and position accuracy — all while keeping the UI responsive. It took considerable trial and error to ensure that each selection properly updated the simulation data and reflected on screen in real time.

The image on the left is the single player simulation player selection screen; on the right is the multiplayer, where you can only choose one team:





5.3 Match Simulation and Scene Reuse

One of the most efficient systems in the game is the simulation viewer. Whether a match is triggered in single-player or multiplayer mode, it is visualized through a shared Unity scene called **SimScreen**. As long as a valid MatchSimulationResult.json file exists in the persistent data path, the simulation will be loaded, parsed, and animated correctly.

This approach allowed me to build a single robust animation and match display system that could handle any match output, regardless of its origin. It also kept the codebase cleaner and avoided the need for maintaining two separate playback systems.

SimScreen uses the MatchAnimator class to read the simulation JSON, spawn player icons on the pitch, and animate their movement frame-by-frame based on the server's iteration logs. The score updates dynamically throughout the match using a timeline of goal events included in the JSON.

5.4 Handling Asynchronous Operations

Unity's coroutine system was used to manage tasks like:

- Fetching player data from the football API
- Posting match data to the Node.js server
- Downloading match results from Firebase Storage
- Animating matches across hundreds or thousands of iterations

Each of these systems had to work independently without blocking the UI or freezing gameplay. One of the more difficult aspects was combining Unity's coroutine model with Firebase's Task-based async methods. I wrote several wrapper systems that allowed me to wait for Firebase responses using coroutines, improving readability and making retry logic easier to implement.

This structure became especially important in multiplayer, where multiple systems needed to wait on one another — such as Firebase database updates, match file availability, and server processing — while still maintaining a seamless experience for the user.

6. Development Process

6.1 Project Planning

The project was structured around several key development phases, including API integration, stat normalization, match simulation, multiplayer functionality, and user interface design. To manage time and maintain direction, I created a biweekly plan, as advised by my supervisor. This approach involved setting specific goals for every two-week period and regularly reviewing whether those goals had been met. The biweekly plan was particularly helpful in identifying when I was starting to fall behind on tasks, and it encouraged me to refocus and reprioritize when unexpected delays occurred —

especially during the more complex multiplayer and simulation debugging phases, as well as times when my other module's assignments began taking up more of my time.

A copy of the biweekly plan is included in the appendix.

6.2 Trello

Since I had previously worked with sprint-based planning during placement and it had been introduced in earlier years of the course, I chose to follow that structure for this project as well. I used Trello to track my sprints, organize features into tasks, and mark items as complete as I progressed (Trello, 2025). It helped me keep a clear overview of what was done, what was in progress, and what still needed attention during each sprint.

6.3 GitHub

I used GitHub throughout development to back up my Unity project and track progress. Commits were made frequently, especially around major feature implementations and bug fixes. I also created a GitHub Pages site associated with the repo, which I used to host a small webpage related to the project.

Because Unity projects include many auto-generated and package-related files, I had to build an extensive .gitignore file to ensure that unnecessary files weren't pushed to the repo. This helped keep the repository clean and made cloning/setup easier. I also included a README.md in the root of the repository to provide an overview of the project, setup instructions, and relevant links.

7. Technical Challenges & Solutions

7.1 Firebase Storage Propagation Timing

One of the earliest issues I encountered with Firebase Storage was the slight delay between uploading a file and it becoming publicly accessible via GetDownloadUrlAsync(). In the multiplayer match flow, this created a problem where the opponent would start checking for the result file immediately after the host uploaded it — but the file wasn't always ready yet, which caused broken download attempts.

To fix this, I removed any fixed retry limits and instead implemented a polling system that simply kept checking every second until the file became available. This change made the system much more reliable and avoided unnecessary user frustration.

7.2 Client Sync and Role Management

Managing player roles between two clients was more complex than expected. Since each user could either be the host (Team One) or the opponent (Team Two), I had to build logic to track each player's role and make sure team selections, match triggering, and result syncing happened in the correct order.

This was handled using a combination of boolean flags and a shared lobby system in Firebase Realtime Database. Making sure the correct team data was written and read by the right user was critical to avoid desyncs or sending invalid match requests.

7.3 UnityWebRequest Failures and Retry Logic

Whether fetching player data, posting to the simulation server, or downloading the match result file, all external communication in Unity relied on UnityWebRequest. During testing, I noticed that small connection issues or server hiccups would sometimes cause requests to fail or return incomplete responses.

To address this, I implemented retry logic with exponential backoff for key network requests. I also wrote custom error logging to make it easier to spot when something went wrong, which helped during both development and testing. These improvements made the entire networking layer of the game more stable and resilient.

7.4 JSON Mapping Edge Cases

Because I was working with external API data and custom match structures, I ran into several issues where the JSON responses didn't exactly match the C# classes I had written. In some cases, players were missing fields, had slightly different key formats, or the nesting didn't deserialize correctly.

Using Newtonsoft.Json made this easier to manage, but I still had to write custom mapping logic and fallback checks, especially in areas where optional or malformed data could break the simulation. This was particularly tricky when converting raw API stats into normalized skills for the simulation engine.

7.5 Debugging Multiplayer Simulation Bugs

Getting the multiplayer simulation system working was one of the most challenging parts of the entire project. The system relied on multiple moving parts — team upload to Firebase, simulation on the server, result upload to Firebase Storage, and download by the opponent — all of which had to happen in the correct order and without conflict.

A lot of time was spent debugging issues where one player's team didn't sync properly, where the simulation result wasn't downloaded, or where the wrong player was assigned to the wrong role. I added detailed logging at every stage of the multiplayer flow, which eventually helped identify and fix issues that weren't obvious just from looking at the UI.

In the end, the system became stable and repeatable, but reaching that point required extensive testing, clear thinking about player state, and frequent adjustments to how the client and server communicated.

8. Conclusions

8.1 Technologies and Tools Used

- Unity for all front-end gameplay, UI, and animations
- C# for logic, player management, and simulation coordination
- Node.js for the custom backend server
- Firebase for authentication, multiplayer lobby sync, and match result storage
- API-Football (API-Sports) to fetch real-world player data and statistics
- GitHub for version control and a public-facing GitHub Pages site
- Postman for API testing and endpoint exploration
- Trello for sprint tracking and progress management
- JSON for all match input/output formatting and data transfer
- AWS and Typescript for my engine's server.

8.2 Self-Review

During the development of this project, I aimed to follow a structured, sprint-based planning system, taking influence from the agile methodologies we had covered in previous years and from my experience during placement. I created a biweekly plan at the outset, setting clear goals for each two-week period. This helped guide my development process and gave me checkpoints to reflect on progress.

However, I found it difficult at times to stick to the original plan. As development progressed, new bugs and unexpected technical challenges — especially around multiplayer syncing, the simulation engine, and player selection logic — often forced me to shift priorities or spend much longer on certain features than expected. These delays were compounded by the demands of other modules and assignments, which sometimes pulled focus away from this project.

As a result, some of the intended features — such as the Leagues tab — were left incomplete by the time of submission. That was disappointing, as I had hoped to include it as part of the full experience. Still, I'm proud of the work I did manage to complete and how far the project came. I stuck to the spirit of my biweekly sprints, even if the tasks themselves weren't always completed in the intended order.

If I were to repeat the process, I would focus more on building in contingency time for debugging and integration work, and be more careful about how shifting one task might impact the progress of others. Despite the setbacks, I'm happy with the final result. The project challenged me technically, pushed me to solve difficult problems independently, and gave me a platform to build something that was tailored for my interests, and which I will continue building and adding on to even after college and the computing expo.

9. Bibliography

API-Football. (2025). *API-Football Documentation*. API-Sports. Available at: https://www.api-football.com/documentation-v3 (Accessed: 29 April 2025).

Firebase. (2025). *Firebase Documentation*. Google. Available at: https://firebase.google.com/docs (Accessed: 29 April 2025).

footballSimulationEngine. (2022). *Football Simulation Engine*. GitHub. Available at: https://github.com/GallagherAiden/footballSimulationEngine] (Accessed: 29 April 2025).

GitHub. (2025). *GitHub Docs*. Available at: https://docs.github.com/ (Accessed: 30 April 2025).

Trello. (2025). *Trello Help*. Atlassian. Available at: https://trello.com/en/guide (Accessed: 0 April 2025).

Postman. (2025). *Postman Learning Center*. Available at: https://learning.postman.com/ (Accessed: 30 April 2025).