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 report outlines the development of my final year project, a mobile football simulation game titled *Fantasy Kickoff*. The game combines a variety of concepts I encountered during my computer science degree — including API integration, data processing, cloud-hosted server architecture, and front-end UI/UX development in Unity — while also allowing me to explore new areas such as multiplayer architecture and client-server data flow.

A central focus of the project was to simulate football matches using real-world player data, rather than fictitious attributes. This design choice informed many of the project's key systems and significantly shaped the development process.

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.
- Implement online multiplayer matches via a cloud-based lobby system.

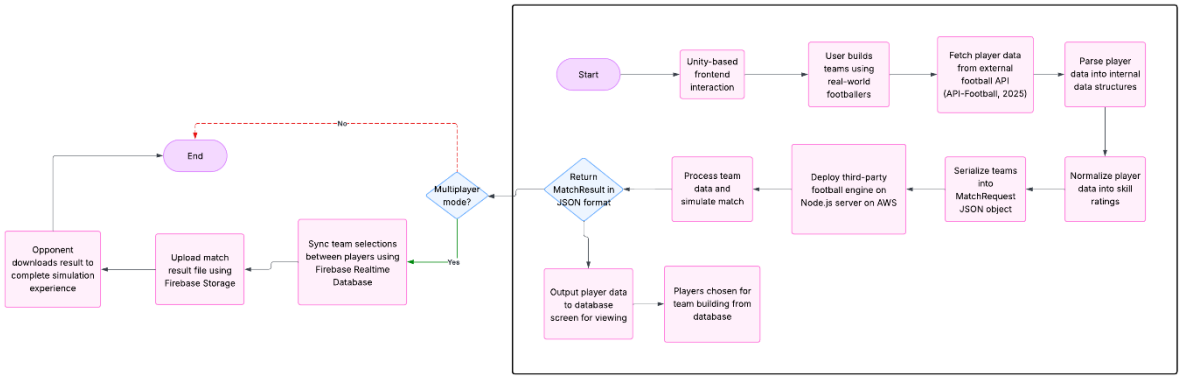
1.3 High-Level System Architecture

The system consists of four main components: a Unity-based frontend (Unity, 2025), an external football statistics API (API-Football, 2025), a custom backend simulation server written in Node.js (Node.js, 2025) and deployed to AWS (AWS Documentation, 2025), and Firebase services for authentication and multiplayer coordination (Firebase, 2025).

Unity is responsible for all player interaction, team selection, and match visualization. Once teams are built using real-world players, they are serialized into a MatchRequest

JSON structure (JSON, 2025), which is then sent to the server. The simulation logic itself is handled by a third-party open-source football engine hosted on GitHub (footballSimulationEngine, 2022). The server processes the match and returns a MatchResult JSON, which is then animated and displayed in the client.

In multiplayer mode, Firebase Realtime Database is used to sync player selections between users, while Firebase Storage handles the match result handoff between the host and opponent.



2. API and Data Processing

2.1 Football API Overview

From the start of the project, one of my core goals was to integrate real-world football data into the game. I wanted the simulation to feel grounded in reality, where player performance was based on live statistics rather than invented attributes. As a result, API research and data integration became the natural starting point.

After evaluating several football APIs, I selected *API-Football*, part of the API-Sports platform, due to its consistent updates, wide coverage of European leagues, and detailed player statistics. It supports a broad range of endpoints — including league data, team rosters, individual player stats, and match records — all accessible via RESTful HTTP requests.

Crucially, the API also included C# support (C#, 2025), which simplified integration into Unity. I used the documentation throughout development to explore endpoints and understand data structures, supported by tools like Postman for testing and prototyping (Postman, 2025).

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

2.2 Fetching and Caching API Data in Unity

API integration was handled using UnityWebRequest to send asynchronous GET requests. These included required headers (like the API key) and query parameters to target specific leagues, seasons, or teams. The data returned was in JSON format, which I parsed using the Newtonsoft.Json library into custom Unity data classes such as PlayerData and TeamData.

This was my first time using UnityWebRequest, and while it required some initial trial and error — particularly in managing headers and async logic — the learning process was rewarding. My previous experience with APIs in a web development context helped accelerate the process, especially when it came to debugging request/response issues and structuring dynamic queries.

To avoid excessive API usage and improve performance, I implemented a local caching system. After a successful API call, the full player dataset was saved as a JSON file to Unity's persistent data path. On future launches, the system checks for this cache before requesting new data, significantly reducing network calls and protecting against API rate limits.

2.3 JSON Parsing into Unity Classes

Once the JSON data was retrieved, I needed to convert it into usable C# objects. Using Newtonsoft.Json, I deserialized the data into custom classes such as PlayerData and Skill, designed to reflect the structure of the API's responses. This allowed me to work with typed objects in Unity, making it easier to display information, link players to teams, and modify data as needed.

At this point, I had built a functioning pipeline for loading, caching, and parsing real-world player data — but I still needed to convert those raw stats into something the simulation engine could work with.

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 Image Downloading and Caching

To improve visual clarity, I also incorporated player headshots into the player list UI. The API provided image URLs alongside each player profile, which I downloaded using UnityWebRequestTexture. Each image was converted into a texture and assigned to the appropriate UI slot.

To avoid downloading the same images repeatedly, I implemented a separate image caching system. Downloaded images were saved as PNGs to persistent storage and checked on future launches. I also configured the cache to clear automatically once per week, ensuring that player images and stats would reflect recent updates over time.

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

2.5 Normalizing and Converting Statistics to Player Skills

The simulation engine required each player to be assigned eight core skills: Passing, Shooting, Tackling, Saving, Agility, Strength, Penalty Taking, and Jumping — all scored on a scale from 1 to 99. These values had to be derived from real-world football statistics provided by the API.

To achieve this, I created a set of formulas that mapped relevant raw statistics to each skill. The goal was to reflect player performance in a way that was both statistically grounded and compatible with the requirements of the simulation engine.

Below is a breakdown of each skill and how it was calculated:

- **Passing**
    - Based on: total passes, key passes, and pass accuracy %
    - Rationale: Total passing volume shows involvement, while key passes indicate creativity. Pass accuracy is weighted to reward consistency.
    - Formula highlights:
        - +1 point for each key pass
        - +2 points for every 5% of pass accuracy
        - Base score starts at 50, adjusted upward

Side note: The simulation showed problems when players with a very low passing score were involved in matches. The matches were low scoring and many of the passes made by players with these low stats made little sense. So, the base 50 passing stat every player gets acts as padding to allow the simulation to function more realistically.

- **Shooting**
    - Based on: total shots, shots on target, goals, shot accuracy, conversion rate
    - Rationale: Combines technical precision with effectiveness. Conversion rate is weighted more heavily than shot accuracy to reflect goal-scoring ability.
    - Formula highlights:
        - rawScore = (accuracy * 0.3 + conversion * 0.7) * 2
        - Weighted by sample size using log10(shotsTotal + 1)
        - Bonus of +5 per 5 goals scored
        - Final score clamped to a max of 100

- **Tackling**
    - Based on: tackles, interceptions, and duel win rate
    - Rationale: Rewards defensive involvement and success in duels. Interceptions are valued equally to tackles, while duel success rate adds consistency.
    - Formula highlights:
        - +2 points per tackle or interception
        - +0.5 points per 1% of duel win rate

- **Saving** *(for goalkeepers)*
    - Based on: saves, goals conceded, and save rate
    - Rationale: Directly reflects the proportion of shots saved versus goals conceded.
    - Formula highlights:
        - saveRate = saves / (saves + conceded)
        - Scaled directly as a percentage to skill value

- **Agility**
    - Based on: dribble attempts, dribble success, and fouls drawn
    - Rationale: Dribbling success reflects agility and movement; fouls drawn imply evasiveness.
    - Formula highlights:
        - dribble success rate + fouls drawn
        - No artificial scaling applied

- **Strength**
    - Based on: duel win rate, fouls committed, and player weight
    - Rationale: Strong players should win more duels and commit fewer fouls. Weight is used as a proxy for physical power.
    - Formula highlights:
        - duel win rate - fouls committed + (weight × 0.5)

- **Penalty Taking**
    - Based on: penalties scored and missed
    - Rationale: Reflects a player's ability to convert penalties under pressure.
    - Formula highlights:
        - penalty rate = scored / total
        - Scaled directly as a percentage

- **Jumping**
    - Based on: player height
    - Rationale: The engine’s original creator also used height as the proxy for jumping ability, due to the lack of actual jump statistics in most datasets. Following this logic, I adopted the same approach. While not a perfect

measure, height gives a reasonable approximation of aerial ability in the context of the simulation.

- Formula highlights:
  - Height (in cm) clamped between 0 and 300

All the player statistics used to calculate these skill numbers are derived from the individual players' performance in the 24/25 season of football, updated every Monday.

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;
```

3. Match Simulation Engine

3.1 Simulation Architecture (Client/Server Split)

The match simulation system implemented in this project is powered by a third-party, open-source football engine sourced from GitHub (GitHub, 2025). Rather than embedding the simulation logic directly into the Unity client, the engine was deployed to a custom Node.js server hosted on AWS . This approach offered a clean separation of responsibilities between the client and server, and provided greater flexibility for handling multiplayer logic, testing, and potential future scaling.

The Unity client is responsible for constructing and sending simulation requests in the form of a structured MatchRequest JSON object. This request contains two full teams of players, each with position data, skill attributes, and formation details. The server receives this payload, runs the simulation using the engine, and returns a MatchResult JSON object containing the outcome of the match.

This architecture enabled consistent and repeatable simulations across both single-player and multiplayer modes. In multiplayer matches, only the host client communicates directly with the server to execute the simulation. The resulting match data is then uploaded to Firebase Storage, allowing the opponent to download the exact same result, ensuring a synchronized experience.

A considerable amount of development time was spent on preparing the data required by the engine. On the client side, this involved enabling users to build custom teams using real-world footballers, assigning them to specific roles and pitch positions through the Unity interface. The PlayerSelectionManager class was developed to manage this process. It calculated precise Vector2 coordinates for each player's starting position based on their assigned role (e.g., ST, CM, CB) and ensured each slot was filled with a valid selection. This strict formatting was critical, as any inconsistency or missing data could cause the simulation engine to crash or produce undefined results.

3.2 MatchRequest & MatchResult JSON Format

The interaction between the Unity client and the simulation server is facilitated entirely through JSON-based communication. When a match is initiated, Unity compiles all relevant data into a MatchRequest object. This includes two teams — each consisting of exactly eleven players — alongside pitch parameters such as width, height, and goal size. These fields are strictly required by the simulation engine's schema and must conform precisely to avoid errors or invalid outcomes.

Each player is also assigned a 2D coordinate on the pitch, determined by internal logic based on their tactical role and formation. For example, defenders are positioned deeper, while attackers are placed closer to the opponent's goal. These positions are calculated using the PlayerSelectionManager and embedded into the MatchRequest. Extensive testing and validation routines were implemented to ensure that all player roles, positions, and skill structures aligned with engine expectations. Invalid or incomplete requests could result in runtime crashes or undefined simulation behaviour, so defensive programming practices were essential.

The request is serialized using the Newtonsoft.Json library and transmitted to the simulation server as an HTTP POST. Upon completion of the simulation, the server returns a MatchResult JSON object. This response contains the match outcome — including goals scored, key player statistics, and optionally, a detailed iteration-by-iteration log of gameplay events. These logs form the basis of the match animation system within Unity and enable frame-accurate playback of each simulation.

This system supports both fixed-team matchups and fully custom teams created through the Unity UI. In single-player mode, the user can manually build two distinct teams and initiate a match between them. The custom selection interface supports full control over team composition, tactical roles, and player attributes. An example of this interface is shown below.

Once the MatchRequest is validated and submitted, it is passed to the backend simulation server, which handles execution and returns the full result for in-game display and analysis.

3.3 Server-side Execution and Logic Flow

The backend simulation server is implemented in Node.js and hosted on an Amazon EC2 instance to ensure reliable, cloud-based access and horizontal scalability (AWS Documentation, 2025). Its main role is to receive structured match data from the client, execute a football match simulation using the third-party engine, and return the result in a structured MatchResult JSON format.

The server exposes a single HTTP POST endpoint at /simulate using the Express.js framework. It accepts a MatchRequest payload containing both teams' data, including player skill values, assigned positions, and pitch configuration. Upon receiving the request, the server parses and validates the data to ensure it conforms to the engine's input structure.

The simulation then proceeds in a loop of discrete time steps, referred to as iterations. Each iteration represents a single unit of simulated time. Within each step, the engine processes player movement, passes, tackles, ball interactions, and scoring events. The match continues for up to 10,000 iterations, with a transition to the second half occurring automatically at iteration 5,000. The engine handles all game logic internally, including transitions between phases like kickoffs, throw-ins, set pieces, and open play.

To enable detailed debugging and animation playback in Unity, I extended the simulation engine to support full per-iteration state logging. This included:

- The 2D position of every player at each iteration (playersOverIterations)
- The position of the ball over time (ballOverIterationsHistory)
- A timeline of score changes with iteration timestamps (scoreTimeline)
- A summary object containing final results, metadata, and team performance (matchDetails)

This information was embedded into the final MatchResult JSON response and returned to the client. Upon receipt, Unity's MatchDataManager class parsed and stored the data locally for later use in match replay and statistical display.

The inclusion of iteration-level logs significantly improved the debugging workflow and enabled more immersive match playback features. It allowed for step-by-step inspection of match flow, tactical positioning, and player decision-making logic. This proved essential for testing player stat influence, refining skill derivation formulas, and ensuring simulation fidelity across both single and multiplayer scenarios.

```
// 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 Node.js simulation server completes the execution of a match and returns a response, the Unity client handles this response via a UnityWebRequest. The HTTP POST call to the /simulate endpoint includes the serialized MatchRequest, and upon successful completion, the body of the response contains a structured MatchResult JSON object.

This JSON payload is parsed immediately upon receipt within the MatchDataManager class. It contains thousands of lines of simulation data — including frame-by-frame position logs for every player (playersOverIterations), ball trajectories (ballOverIterationsHistory), real-time score updates (scoreTimeline), and a final summary object (matchDetails) that aggregates goals, stats, and overall outcomes.

To ensure the data can be accessed persistently and reused offline, the full JSON string is written to a local file using File.WriteAllText(), stored in Unity's Application.persistentDataPath. This file acts as the primary source for replaying the simulation and visualizing the match in the next phase of the user experience.

In multiplayer mode, only the host client triggers the simulation. Once the result is received, the JSON file is uploaded to Firebase Storage. The opponent client continually

polls Firebase until the file becomes available, then downloads it using UnityWebRequest.Get() and saves it locally in the same format. This ensures that both clients share a single source of truth for the match, preserving simulation integrity and preventing any discrepancies between the two players' views.

Immediately after the file is saved locally, the game transitions to the match playback scene (SimScreen). This scene loads the saved MatchResult file and begins animating the match from start to finish based entirely on the data received from the server .

3.5 Match Playback and Animation

Once the MatchResult.json file is received from the server and saved locally, the Unity application transitions to the simulation playback scene (SimScreen). This scene is managed by the MatchAnimator class, which is responsible for loading, parsing, and visualising the entire match simulation frame-by-frame based on the per-iteration data provided by the engine.

At runtime, the MatchAnimator class loads the saved MatchSimulationResult JSON file and deserializes it into structured C# objects. This data includes:

- A list of every player's position over time (playersOverIterations)
- The full trajectory of the ball (ballOverIterationsHistory)
- A timeline of score updates (scoreTimeline)
- Match metadata (matchDetails), including pitch layout and total iteration count

The system then instantiates Unity UI elements for each player and the ball using preconfigured prefabs. Each player is assigned a GameObject represented as a colored dot — red for the kickoff team and blue for the second team — and their initial positions are derived from the first recorded iteration.

To animate movement, each player and the ball is managed by a corresponding internal class: AnimatedPlayer or AnimatedBall. These classes track the object's position at each simulation tick and interpolate their movement using Unity's Vector2.Lerp() function, providing smooth transitions between positions over time. Movement is updated in real-time during a coroutine (PlayMatch()), which steps through up to 10,000 iterations, updating positions and match state on each frame.

A key feature of this system is the use of lerpProgress and position interpolation to simulate continuous motion, even though the engine records only discrete positions at each iteration. This allows for smoother, more readable playback and helps recreate the flow of a real football match.

In parallel, the system tracks the match score. At each iteration, the UpdateScore() function checks the current index in the scoreTimeline array. If a new goal event has occurred, the relevant score text (kickoffTeamScoreText, secondTeamScoreText) is

updated accordingly. This ensures the visual score aligns perfectly with what occurred during the actual simulation.

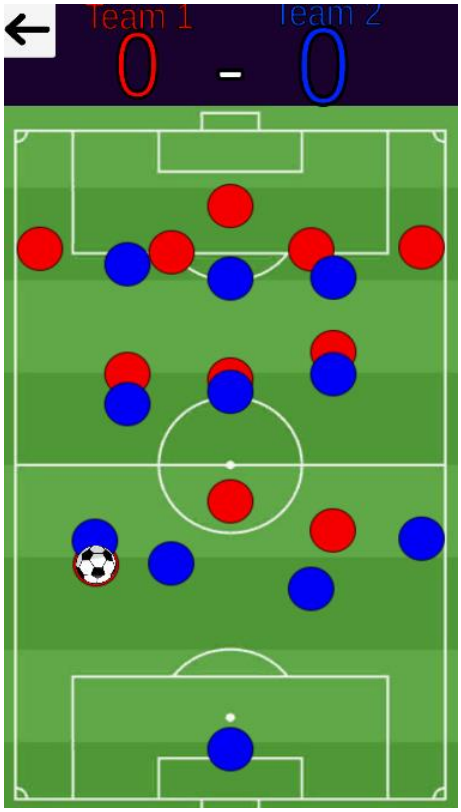
All positions are centered using the screen's pitch dimensions to match the Unity canvas layout with the coordinate system used by the engine. This transformation ensures accurate placement on the UI regardless of screen resolution.

In total, the playback system provides:

- Real-time animation of both teams based on engine simulation
- Accurate visualisation of ball movement
- Dynamic score updates tied to real goal events
- A smooth and uninterrupted recreation of the simulated match

This system allows users to not only see the result of the match numerically but experience it as a full visual replay — an essential part of making the simulation feel dynamic and engaging. It also lays the foundation for more advanced visual enhancements in the future, such as player labels, possession indicators, or key moment replays.

A still of the match animation:



With the footballer database, stat conversion system, and match simulation pipeline in place, the next major goal was to expand beyond a purely single-player experience. To support multiplayer functionality — and to lay the groundwork for potential future features such as player profiles, stat tracking, or persistent customisation —

implementing user authentication became the next development priority. This enabled a secure and structured system for managing users, creating match lobbies, and facilitating competitive matches between players.

4. Multiplayer System with Firebase

4.1 Firebase Authentication and Lobby System

In this project, Firebase Authentication was used to manage user sign-in, identity, and session continuity. Players could register or log in using their email and password. Upon successful authentication, Firebase generated a unique UID (user identifier) for each player, which was then used to associate their multiplayer actions — such as team selection, lobby creation, and match participation — with their identity across sessions.

Once authenticated, users were brought to a multiplayer interface that allowed them to either host a new match or join an existing one using a unique lobby ID. The lobby system was implemented using Firebase Realtime Database. Each lobby was stored as a node in the database under the /lobbies path, with the lobby ID as the key. Inside each lobby node, subfields were used to store structured data, including:

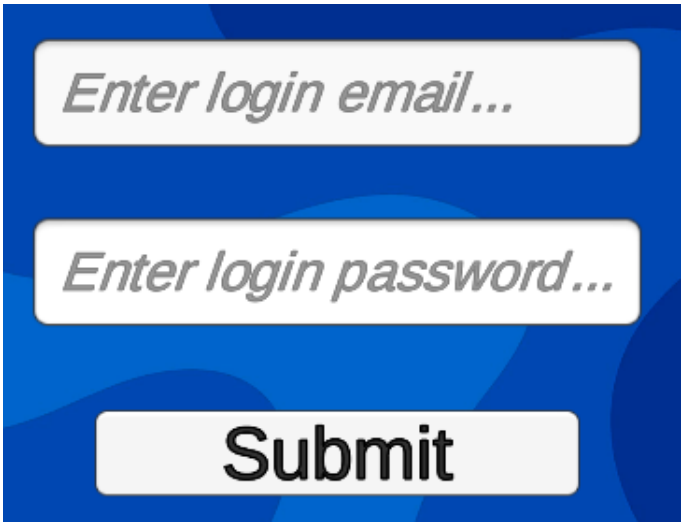
- hostUID: The UID of the player hosting the match
- teamOne and teamTwo: Serialized JSON strings representing each player's selected team
- matchResult: An optional field added once the simulation was complete (handled later)

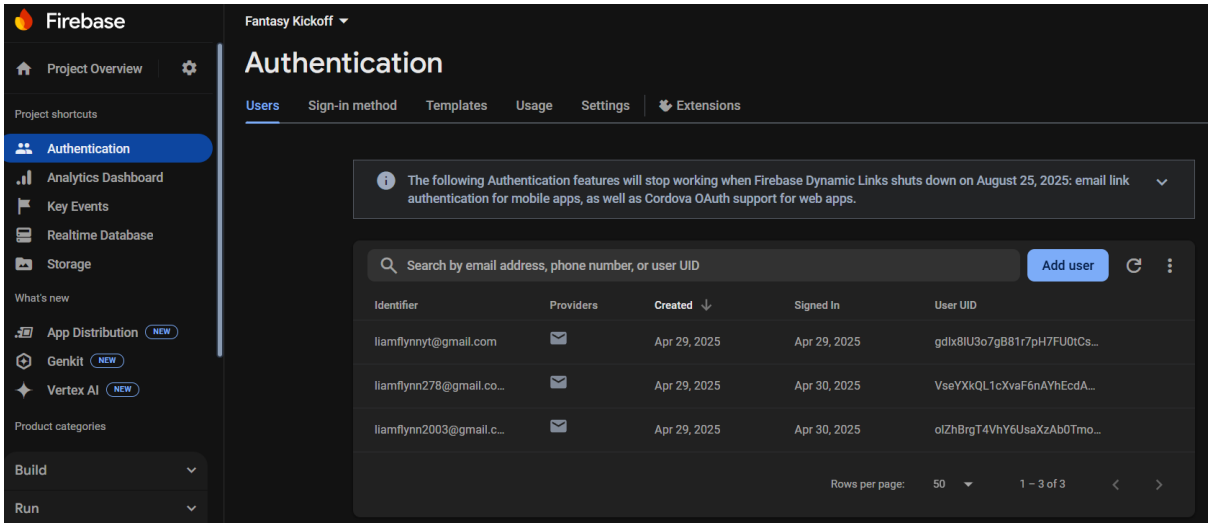
Firebase's real-time capabilities allowed the application to listen for updates to the lobby node. This meant that once both players had submitted their teams to the database, the host client could immediately detect this and trigger the simulation. This architecture enabled peer-to-peer coordination without requiring traditional networking stacks like WebSockets or direct client-to-client communication.

In addition to handling real-time data sync, Firebase Authentication also played a role in future extensibility. By linking each match interaction to a persistent user account, the system could be expanded to include user profiles, win/loss history, saved teams, or friend lists — all without needing to build a separate identity management backend.

This integration of Firebase allowed for a secure, scalable, and reliable foundation for the multiplayer layer of the game, while remaining lightweight and Unity-friendly for mobile deployment. It was also used in the lobby system, which was created to facilitate multiplayer matches.

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





4.2 Multiplayer Lobby Flow

After signing in, users can access the Multiplayer Quickmatch section via the bottom navigation bar. From there, they are given two options: to create a new lobby or join an existing one using a six-digit lobby code.

When a player chooses to create a lobby, the system generates a unique alphanumeric code and stores a new node in Firebase Realtime Database under the /lobbies path. This node includes metadata such as the creator's UID, the timestamp of creation, and placeholder fields for both teams. The generated code is displayed to the host and can be shared with a friend, allowing them to join the lobby.

Players joining a lobby enter the code provided by the host. The system then checks for the existence of a corresponding node in the database. If the lobby exists and has not yet been filled, the joining player is connected and assigned as the second user. Once both players are present, the interface transitions to the team selection screen, where each user can independently choose their starting eleven.

This lobby system provides a lightweight yet effective alternative to traditional matchmaking or socket-based networking. It allows two users to connect and compete without requiring simultaneous login or constant polling. The use of Firebase ensures real-time updates and handles state synchronization between both clients.

From a technical perspective, this system is resilient to delays and connection variability. Users can create or join lobbies at any time, and the database acts as a central source of truth, maintaining lobby state even if one user is temporarily disconnected. This design also lays the foundation for future enhancements such as lobby expiration, player rematches, or match history tracking.

4.3 Realtime Team Data Sync (Database)

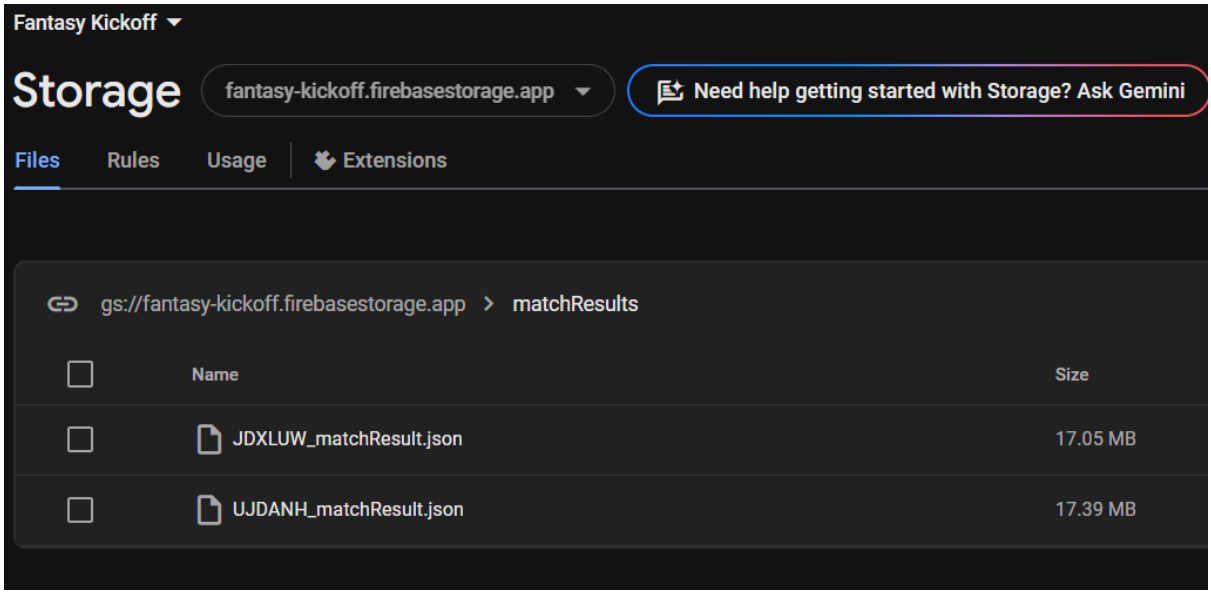
Once both players are connected to a shared lobby, each is presented with the familiar team selection interface used in single-player mode. After selecting their teams, the players submit their chosen lineups to Firebase Realtime Database.

Each team is serialized into JSON using the same internal structure defined for the simulation engine. This includes player names, positions, skill values, and current pitch coordinates. The data is then uploaded to the appropriate key within the lobby node — either teamOne or teamTwo, depending on whether the user is the host or the opponent.

Firebase's real-time database capabilities are leveraged using asynchronous listeners attached to the lobby node. These listeners are triggered when both teamOne and teamTwo have been uploaded, at which point the host automatically initiates the match simulation. The opponent simply waits for the match result to be uploaded to Firebase Storage, at which point they retrieve and view the same simulation outcome.

Adapting the single-player selection system for multiplayer use presented several challenges. Not only did the UI need to accommodate role-based logic (e.g., who is Team One vs. Team Two), but the entire player selection flow had to be validated and serialized deterministically. Mistakes such as missing player data or mismatched formation structures could cause the server-side simulation to fail. To mitigate this, I implemented strict checks prior to submission and ensured that each player's data was properly validated and complete before the match could begin.

This cloud-based syncing model worked reliably and avoided the need for real-time socket connections. By using Firebase's structured JSON and listener system, I was able to create a seamless team-building experience that felt responsive while maintaining consistent state across both clients.

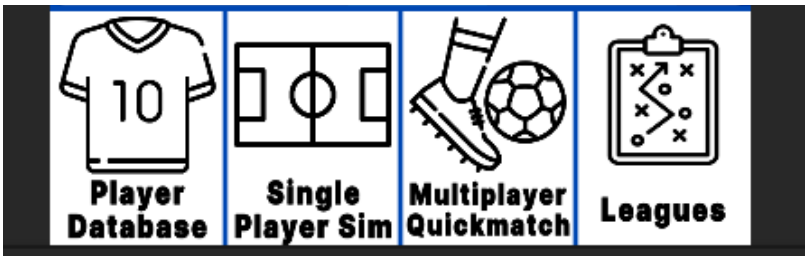


5. Unity Scenes and UI

5.1 Player Selection UI

Upon launching the application, users are presented with an initial homescreen that displays the game's logo and options to either sign up or log in. Authentication is handled via Firebase Authentication, and upon successful login, users are redirected to the Main Hub — a centralised interface that provides access to all gameplay functionality.

The user interface within the main hub is designed around a single-scene architecture, where navigation between core features is handled through a persistent bottom navigation bar. This approach avoids unnecessary scene loading, reduces memory usage, and improves UI responsiveness — particularly important for mobile deployment.



The navigation bar provides access to the following four primary sections:

Player Database – A scrollable, searchable list of all footballers currently available in-game. Each entry includes the player's image, name, skill rating, and position, populated from cached API data.

Single-Player Simulation – A mode where users can build two custom teams from available players and simulate a match between them using the same backend system as multiplayer mode.

Multiplayer Quickmatch – A system allowing users to either create or join a Firebase-backed lobby and compete against another player in a simulated match.

Leagues (Planned Feature) – A placeholder for a future feature designed to allow users to browse league-level statistics and simulate structured tournaments. Although not implemented in the current version, the UI placeholder and navigation logic are in place, and the feature is planned for completion ahead of the project's public expo.

Each of these sections is rendered within the same Unity scene. Transitions between them are handled through UI state toggling, using CanvasGroups and GameObject activation/deactivation. This decision significantly improved runtime performance and reduced UI latency by preventing costly scene transitions and enabling persistent data access across tabs.

The player selection system itself is modular and re-used across both single-player and multiplayer modes. Player selection slots are visually linked to tactical positions and are populated through Unity UI components bound to underlying data structures managed by the PlayerSelectionManager. This integration ensures that any updates to the player database or UI logic are automatically reflected across both game modes.

5.2 Player Selection and Team Building

Team selection is a core system shared between both single-player and multiplayer modes. Users are presented with a set of UI slots representing fixed football roles (e.g., GK, CB, ST), and can populate each slot by selecting a footballer from the real-world player database.

Each selection updates only the player's name in the corresponding UI slot, providing immediate visual feedback to the user. Behind the scenes, however, this interaction triggers more complex updates to the internal data structures. The logic is managed by the PlayerSelectionManager script, which maps each slot to:

- A positional label (e.g., "RB", "LW")
- A starting pitch coordinate, expressed as a Vector2, used by the simulation engine

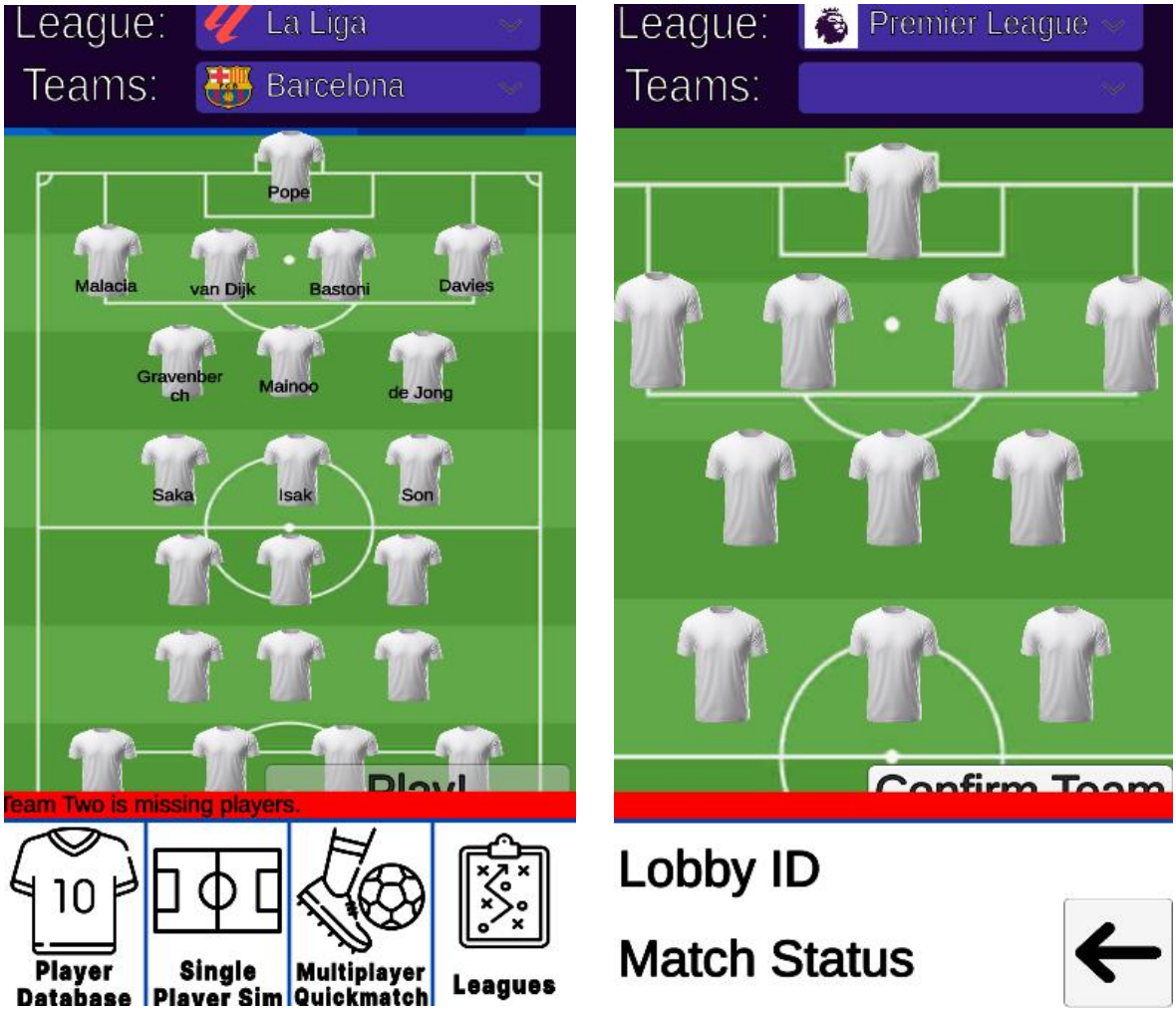
This information is essential for constructing the MatchRequest JSON that is sent to the simulation server. Each selected player must be assigned a valid position and coordinate, as well as associated skill data. Any mismatch or incomplete configuration could cause the simulation to fail or return undefined results. To prevent this, the selection system includes validation checks that ensure team slots are fully and correctly filled before allowing progression.

In multiplayer mode, only one team can be selected — either Team One (host) or Team Two (opponent) — based on the user's role in the Firebase lobby. The selection UI adapts accordingly, disabling or hiding the second team's selection grid.

The overall system required careful design to ensure that it worked consistently across both modes, maintained valid data structures, and provided responsive updates to the UI. Although only the player's name is displayed on-screen, all associated data is stored and serialized for use in the backend simulation.

The image on the left shows the team selection interface in single-player mode, where users build both teams. The image on the right displays the multiplayer variant, where only one team is selectable.

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: SimScreen

A key architectural decision in this project was the design of a unified playback system that could support both single-player and multiplayer simulations. To achieve this, a single dedicated Unity scene — named **SimScreen** — was created to handle all match visualisation. Regardless of the game mode, once the match simulation is complete, the application transitions to this shared scene to display the results.

The **SimScreen** scene operates independently of how the simulation was triggered. Its functionality is entirely driven by the existence of a local file named MatchSimulationResult.json, which is written to Unity's Application.persistentDataPath. As long as this file is valid and well-formed, the scene loads and animates the match consistently. This decoupling of the simulation logic from the animation logic allows for flexibility and reduces the risk of mode-specific bugs.

The scene is managed by the MatchAnimator class, which handles the following responsibilities:

Parsing the match result: The JSON file is deserialized into C# classes representing players, ball states, iteration histories, and score timelines.

Instantiating player GameObjects: Each player is spawned as a UI element and color-coded based on their team.

Animating movement: The system interpolates positions frame-by-frame using per-iteration data captured during simulation.

Updating score: Goals are tracked using a timeline array (scoreTimeline) and the UI is updated at the correct simulation frame.

The reusability of this scene is made possible by ensuring that the input (the JSON file) is always formatted according to a strict schema, regardless of whether the simulation was triggered by the single-player or multiplayer system. This abstraction provides multiple advantages:

Scalability: Any improvements made to the playback system (e.g., improved visuals or player metadata) automatically benefit both game modes.

Reliability: The scene's operation is relies solely on pre-generated data rather than runtime simulation, which simplifies debugging.

This design also opens the door for future use cases, such as:

Replays from stored matches

Match sharing between users

Exporting match videos for highlights or analytics

By abstracting playback into a single, reusable component, a cleaner architecture and a smoother, more unified user experience was created. The development time that went into each individual feature and scene varied, with some being more problematic to implement than others.

6. Technical Challenges & Solutions

6.1 Firebase Storage Propagation Timing

During multiplayer matches, the host uploads the MatchResult.json file to Firebase Storage after completing the simulation. However, Firebase Storage occasionally introduces a brief delay between the completion of a successful upload and the point at which that file becomes publicly accessible via GetDownloadUrlAsync().

Initially, this caused a problem where the opponent would immediately begin polling for the match file, resulting in download attempts that failed due to the file not yet being available. This inconsistency introduced unnecessary user confusion and broke the smooth flow of the match reveal.

To address this, I replaced fixed retry logic with a continuous polling coroutine. The system now repeatedly requests the download URL at one-second intervals until it becomes valid. This approach greatly increased the robustness of the multiplayer flow, ensuring that players never experience failed downloads due to propagation lag. Additionally, error messages were suppressed during polling to avoid false error flags in the UI.

6.2 Client Sync and Role Management

One of the most critical aspects of multiplayer functionality was ensuring correct client state synchronisation. Each lobby consists of two users — one hosting (Team One) and one joining (Team Two). It was essential to ensure that the right team data was uploaded and downloaded by the correct player, and that no conflicts occurred during team selection or match simulation.

To manage this, I used Firebase Realtime Database to store lobby nodes with clear role definitions (hostUID) and designated team slots (teamOne, teamTwo). Each client determined its role based on UID comparison and used this to:

- Decide which UI to display (single team vs. dual-team selector)
- Upload to the correct database key
- Handle match result download logic accordingly

A boolean flag system on the client side tracked player roles and was used throughout the match flow to route logic correctly. Role misassignment would have led to team mismatches or replaying the wrong simulation — an issue I debugged thoroughly using UID logs and structured logging during Firebase operations.

6.3 UnityWebRequest Failures and Retry Logic

Networking in Unity was handled using UnityWebRequest for all external API communication, including:

- Fetching football data from the API
- Posting match requests to the simulation server
- Downloading match results from Firebase Storage

Throughout testing, I observed that requests occasionally failed due to poor connectivity or temporary server downtime. These failures caused broken gameplay sequences or silent hangs in earlier builds.

To solve this, I implemented a retry mechanism using exponential backoff on critical requests. For example, when posting to the simulation server, if the request failed, the client would wait and retry up to a maximum number of times. For Firebase downloads, polling was used (as described in Section 6.1).

Additionally, I wrote a set of custom logging tools to expose request states in the Unity console, including error codes, failure points, and timeouts. These logs were vital for diagnosing edge-case errors and allowed me to tune retry thresholds for reliability without creating infinite loops or excessive delays.

6.4 JSON Mapping Edge Cases

The game relied heavily on structured JSON to transfer player and match data between systems. This introduced challenges, particularly when parsing external data from the football API, where fields were occasionally missing, malformed, or nested unpredictably.

For example:

- Some players were missing subfields like pass.accuracy or duels.won
- Others had data formatted as strings instead of numbers, depending on the API source

I used Newtonsoft.Json for deserialization but had to write custom mapping logic and apply fallback defaults for incomplete data. For instance, if a stat like shots.on was null, it defaulted to zero to avoid calculation errors in skill derivation.

These edge cases also appeared during match simulation output. As I added support for per-iteration data (playersOverIterations, ballOverIterationsHistory), I had to ensure Unity's JSON parser could handle long arrays and preserve float precision. I implemented validation checks to test for common deserialization errors and used Unity’s built-in tools to verify structural consistency during loading.

6.5 Debugging Multiplayer Simulation Bugs

The most complex part of the entire system was the multiplayer simulation flow. It relied on a tightly ordered sequence of events across both clients:

1. Both users select their teams
2. Teams are uploaded to Firebase
3. Host triggers match simulation
4. Result is uploaded to Firebase Storage
5. Opponent downloads the file and starts animation

Any deviation — such as one user uploading an incomplete team, simulation being triggered too early, or Firebase delays — could result in a failed match.

To debug these scenarios, I implemented extensive logging at each phase of the multiplayer lifecycle, including:

- Team submission status
- Role assignment verification
- MatchResult presence
- Firebase download state

These logs were displayed in Unity's developer console and used color-coded messages to highlight logic paths. Over time, this system helped me isolate and resolve issues such as missing team keys, incorrect JSON formatting, and premature simulation triggers.

7Development Process

7.1 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.

7.2 Trello

Given my prior exposure to sprint-based development workflows during industry placement and earlier coursework, I adopted a similar agile-inspired approach for this project. I used Trello, a lightweight kanban-style task management tool, to structure and monitor development progress across biweekly sprints (Trello, 2025).

Each sprint was broken down into granular tasks and feature goals, which were organized into columns such as "To Do," "In Progress," and "Completed." This allowed for clear visualisation of the current development status at any point in time and helped prioritise critical tasks, such as integrating multiplayer logic or debugging the simulation engine.

Trello's flexibility made it well-suited for managing solo development. It allowed me to dynamically adjust task priority, add notes or blockers to cards, and link related components such as script files or assets. While not integrated into version control directly, it complemented GitHub well by acting as a higher-level planning layer above the codebase.

Using Trello contributed to better task isolation and reduced the risk of overlooking smaller dependencies during more complex phases of the project. A screenshot of the Trello board is included in the appendix for reference.

7.3 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.

8. Conclusions

8.1 Technologies and Tools Used

- Unity – Front-end engine used for gameplay, UI rendering, and simulation animation
- C# – Core programming language for logic implementation, data structures, and system coordination
- Node.js – Backend server used to host the simulation engine and process match logic
- Firebase – Used for authentication, multiplayer lobby management, real-time database sync, and file storage
- API-Football (API-Sports) – External API used to retrieve real-world football data and player statistics
- GitHub – Version control platform used throughout development, including hosting a GitHub Pages site
- Postman – API testing tool used to validate endpoints and examine JSON structures
- Trello – Sprint planning and task tracking across biweekly development cycles
- JSON – Core data format used for communication between Unity and the simulation server
- AWS & TypeScript – Used to deploy and customise the simulation engine server infrastructure, (TypeScript, 2025)

8.2 Self-Review

This project was an opportunity to apply a wide range of skills developed over the course of my degree, while also exploring new tools and workflows. From the outset, I followed a sprint-based planning model inspired by agile practices introduced during previous modules and refined during my industry placement. I developed a biweekly plan to track goals, measure progress, and structure the work into manageable phases.

Despite these efforts, the actual development process was more complex than anticipated. Technical challenges — particularly around multiplayer synchronisation, simulation accuracy, and dynamic team selection — often forced me to re-prioritise tasks and allocate more time to unexpected issues. These delays, combined with deadlines from other modules, meant that not every planned feature was completed. One such example was the Leagues tab, which remains a work-in-progress.

That said, I am proud of the features I was able to deliver: a working end-to-end simulation pipeline powered by real-world data, a multiplayer match system with Firebase integration, and a clean, reusable animation system. I believe the overall

structure of the game is strong, and I've built a platform that I can continue to expand — both for the Computing Expo and beyond graduation.

If I were to approach this again, I would allow for more buffer time for debugging, better anticipate integration overhead between systems, and consider locking down the scope earlier. Nonetheless, the project pushed me technically, encouraged independent problem-solving, and allowed me to create something aligned with both my academic background and personal interests — a football simulation system I'm excited to keep improving.

9. Bibliography

API-Football. (2025) *API-Football Documentation*. API-Sports. Available at: https://www.api-football.com/documentation-v3 (Accessed: 30 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).

Unity Technologies. (2025). *Unity Manual*. Available at: https://unity.com (Accessed: 30 April 2025).

C#. (2025). *C# Documentation*. Microsoft. Available at: https://learn.microsoft.com/en-us/dotnet/csharp/ (Accessed: 30 April 2025).

Node.js. (2025). *Node.js Documentation*. OpenJS Foundation. Available at: https://nodejs.org/en/docs (Accessed: 30 April 2025).

AWS. (2025). *AWS Documentation*. Amazon Web Services. Available at: https://docs.aws.amazon.com/ (Accessed: 30 April 2025).

TypeScript. (2025). *TypeScript Documentation*. Microsoft. Available at: https://www.typescriptlang.org/docs/ (Accessed: 30 April 2025).

JSON. (2025). *Introducing JSON*. JSON.org. Available at: https://www.json.org/json-en.html (Accessed: 30 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: 30 April 2025).

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