LITHAN

UDM CLaaS®

Project Report

Qualification Name	BDSE
Module Name	CAI

Student name	**Assessor name**
Escarro, Sebastian Seth R.	

Date issued	**Completion date**	**Submitted on**
1/17/26		

Project title	**Capstone Software Development Project**

Learner declaration

I certify that the work submitted for this assignment is my own and research sources are fully acknowledged.

Student signature: 

Date: 1/17/26

LITHAN

EDUCLAAS®

Index

Document Version History

Version Number	Effective Date of release	Details	Author
1.0	15 Nov 2025	Initial Creation	

Project Overview

Introduction

Maimai is an arcade rhythm game developed by sega. It's a game where you tap buttons or the screen to the beat of the music. There are plenty of people who would visit their local arcades in order to experience this game. They have developed a unique "rotation" culture where players wait their turn to play a set number of songs. Currently, this process is manual, relying on physical queues or improvised lists. This leads to uncertainty regarding wait times and difficulty for solo players to find partners for "Sync" (multiplayer) gameplay. As arcade traffic fluctuates, players waste significant time waiting or miss their turn entirely. **MaiQueue** is a cross-platform web application designed to optimize the queuing experience for Maimai. It replaces disorganized physical lines with a digital, real-time queuing system. The system supports multi-branch locations (e.g., SM Seaside, SM Jmall), hybrid user access (Registered vs. Guest), and provides real-time wait list visibility to reduce congestion.

Project Objectives

General Objective: To design and develop a full-stack web application using React and Python that digitizes the queueing process for *maimai* and integrates Artificial Intelligence to optimize user experience.

Specific Objectives:

1. To implement a queue system for arcade rhythm game maimai
2. To develop a Predictive Wait-Time Module using Random Forest Regression that estimates wait times based on queue length, day of the week, and historical session duration (avg. 15 mins).
3. To create a Smart Matchmaking System using K-Nearest Neighbors (KNN) to pair solo players based on their DX Rating and preferred difficulty level.

Scope of the Project

This study focuses on the development of **"MaiQueue,"** a web-based queue management application designed specifically for the arcade rhythm game *maimai*. The primary goal is to digitize the manual "rotation" system used in local arcades.

The project capabilities are limited to the following:

- **Target Users:** The system is designed for arcade players who wish to queue for *maimai* machines and finding gameplay partners.
- **Platform:** The application will be a mobile-responsive web application accessible via standard mobile browsers (Chrome/Safari).
- **Real-Time Queue Management:** Utilization of **Firebase Firestore** to provide instantaneous updates of the queue list, allowing users to join, leave, and view their position in real-time without refreshing the page.
- **AI-Powered Matchmaking:** Implementation of a **K-Nearest Neighbors (KNN)** algorithm using Python (Scikit-Learn). This module will analyze a user's self-reported "DX Rating" to suggest optimal partners within the current active queue.
- **Wait-Time Estimation:** Implementation of a **Linear Regression** model that calculates estimated wait times based on the number of active players, the number of available cabinets, and a fixed average session duration (15 minutes).

- **User Accounts:** Basic authentication via **Firebase Auth** (Google or Email/Password) to track user history and queue status.

Limitations of the Study

To ensure the feasibility of the project within the given timeframe, the following constraints are set:

- **No Hardware Integration:** The system will **not** utilize external sensors, cameras, or IoT devices to detect machine occupancy. All status updates (e.g., "Game Started", "Game Finished") rely on manual user input within the web app.

- **No Direct API Access:** The application is **not** connected to the official Sega servers or *maimai DX NET*. It does not fetch real-time game data. All player statistics (Rating/Class) are **self-reported** by the user and are not automatically verified by the system.

- **User Compliance Dependency:** The accuracy of the queue and wait-time predictions relies heavily on users honestly checking in and checking out. The system cannot prevent "ghosting" (users leaving the arcade without leaving the digital queue) beyond a manual timeout function.

- **Internet Connectivity:** As a cloud-based application, the system requires a continuous internet connection to function. It does not have an offline mode.

- **Scope of Matchmaking:** The matchmaking algorithm is limited to numerical rating proximity. It does not account for complex player preferences such as specific song genres or "Accuracy vs. Score" playstyles.

Methodology

The project development followed the **Agile Software Development Methodology**, specifically utilizing an iterative and incremental approach. This method was selected to accommodate the complexity of integrating Artificial Intelligence with a real-time web application.

The development lifecycle was divided into three distinct sprints:

- **Sprint 1:** Focused on establishing the **Serverless Architecture** using Firebase and React. This phase prioritized the creation of the database schema (Users, Queues, Branches), the basic user authentication flows, and connecting the React frontend to the Python backend endpoints,
- **Sprint 2 :** Focused on the "Mobile-First" User Interface design, building the actual queue logic, and refining the real-time data synchronization listeners.
- **Sprint 3:** Dedicated to the Backend API development. This involved setting up the Python environment and engineering the **Linear Regression** and **K-Nearest Neighbors (KNN)** algorithms using mock data to ensure statistical validity.

Background and Problem Statement

Context and motivation

This project was imagined out of frustration from arcade queues. As an avid player of this game, I've experienced how awful these queues could be. While not always, when it's bad, it's bad. This project has been in the back burner for me but I simply didn't have the time to actually start with. This has given me the opportunity to actually start working on it.

Problem description

The study aims to address the inefficiencies in the current manual queueing system for *maimai* players. Specifically, it seeks to answer the following:

1. **Unpredictable Wait Times:** Players cannot accurately estimate when their turn will be, leading to loitering or missed turns.

2. **Inefficient Matchmaking:** Solo players struggle to find partners of similar skill levels for dual-play (Sync) modes.

3. **Lack of Remote Monitoring:** Players must be physically present to check queue status, preventing them from utilizing wait time for other activities.

4. **Queue "Ghosting":** Manual lists often contain names of players who have already left, stalling the line.

Assumptions

- **Single-Device Integrity:** It is assumed that each player uses a single mobile device to queue. The system does not currently prevent a single user from logging into multiple devices to manipulate the queue (though account restrictions exist).
- **Operational Consistency:** The system assumes that the physical arcade machines are operational unless an Admin manually updates the cabinetCount in the Branch settings.
- **Standard Playtime Adherence:** The wait-time algorithm assumes that players will generally adhere to the "rotation culture" etiquette (approx. 15-20 minutes per turn). It treats outliers (players taking 40+ minutes) as anomalies to be filtered out.
- **Browser Compatibility:** It is assumed users are accessing the application via modern mobile browsers (Chrome, Safari, or Firefox) that support JavaScript ES6+ and WebSocket connections required for Firestore.

Project Proposal and Planning

Timeline

Phase	Dates	Deliverables
Week 1: The Skeleton	**Jan 3 – Jan 9**	**Setup & Database** • Setup React + Flask/FastAPI. • Create Database (Users, Queue Tables). • **Goal:** A website where you can click "Join" and your name appears in a list.
Week 2: The Brains	**Jan 10 – Jan 16**	**AI Implementation** • **Matchmaking:** Write the KNN script (Python) to find similar numbers. • **Wait Time:** Write the math formula for the timer. • **Goal:** The website tells you "Wait: 15 mins" and "Matched with: Player B".
Week 3: The Face	**Jan 17 – Jan 22**	**UI Polish & Connection** • Make the React buttons look nice. • Ensure the Frontend talks to the Backend correctly. • **Goal:** A working demo video.
Final Prep	**Jan 23 – Jan 24**	**Documentation & Defense** • Write the paper. • Create the slide deck. • **Goal:** Submission.

Stakeholders

- **Primary Stakeholders:** The Maimai Players who directly interact with the application to join queues, find partners, and track their turn. They benefit from reduced loitering time and improved social matching.
- **Secondary Stakeholders:** The Arcade Managers or Community Leaders who have "Admin Mode" access. They are responsible for managing the queue order (e.g., removing "ghost" players)
- **Tertiary Stakeholders:** Arcade Venues (e.g., SM Seaside, SM Jmall), which benefit indirectly from organized crowds and potentially higher machine utilization due to efficient queuing.

Tools and Technology

Category	Technology	Purpose
Front End	React.js	User interface for the queue board and joining/leaving.
	Tailwind CSSS	Fast styling to ensure the app looks good on mobile phones.
Back End	Python	A lightweight server specifically to run the AI algorithms.
	Scikit-Learn	The library used for the **Wait Time Regression** and **KNN Matchmaking**.
Database & Auth	Firebase Firestore	A NoSQL cloud database that handles real-time queue synchronization.
	Firebase Auth	Manages user sign-ins (Google/Email) without needing to write custom security code.
Deployment	Vercel / Render	Free platforms to host the website and backend for the demo.

System Design and Architecture

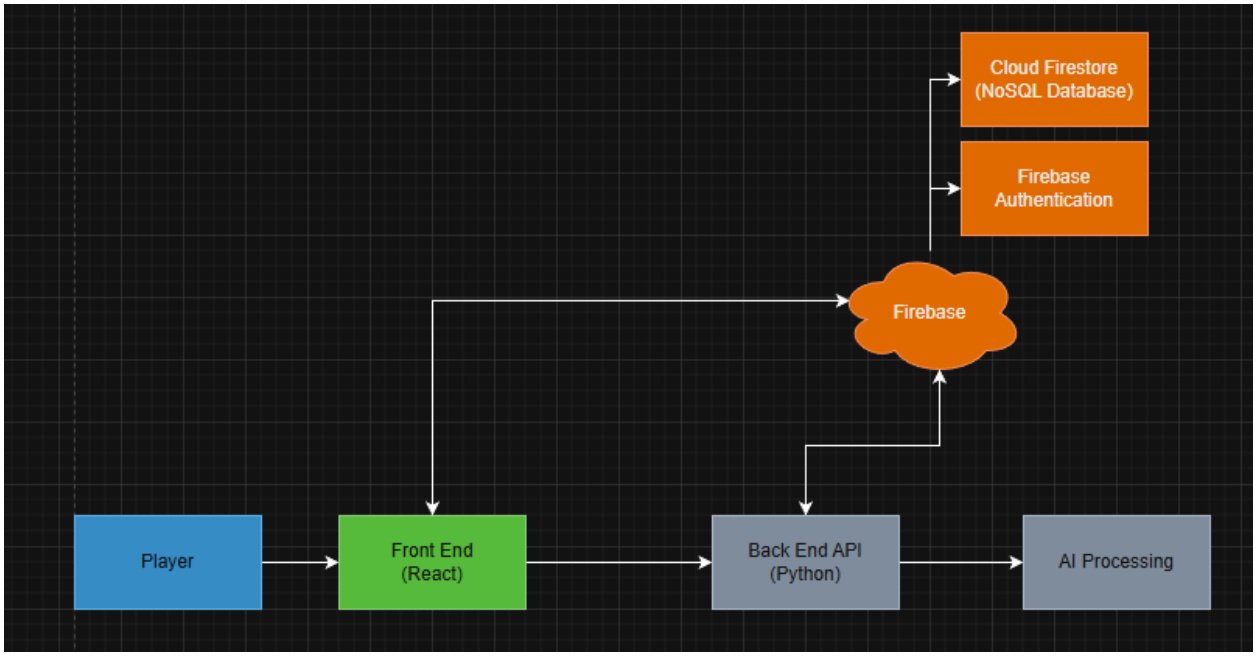
The system follows a **Serverless Architecture** pattern, leveraging Firebase Backend-as-a-Service (BaaS) to ensure high availability and real-time data synchronization without managing physical servers.

High-Level Architecture Block Diagram

The diagram below illustrates the high-level data flow. The **Front End (React)** acts as the central interface, communicating with Firebase for real-time data (**Cloud Firestore**) and Identity Management (**Firebase Authentication**). A dedicated **Back End API (Python)** handles complex AI processing tasks and feeds this intelligence back into the Firebase ecosystem.

Key Components:

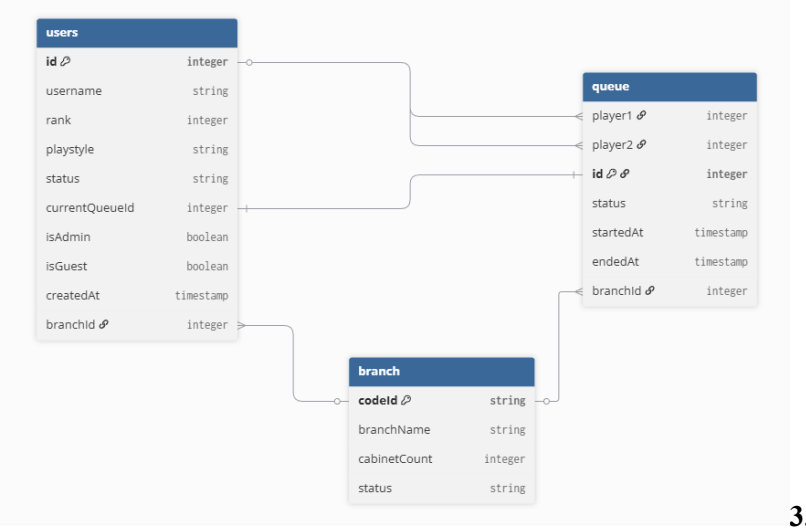
- **Player (User):** Interacts with the system via a web browser on a mobile device.
- **Front End (React):** The reactive user interface that displays the queue and handles user inputs.
- **Firebase Ecosystem:**
    - **Cloud Firestore:** A NoSQL database storing user profiles, queue sessions, and branch data.
    - **Firebase Authentication:** Manages secure logins for both Google users and Anonymous guests.
- **Back End API (Python):** An external service that processes historical session data to generate AI insights (e.g., wait time predictions).



Database Design (Data Model)

The database utilizes **Cloud Firestore**, a NoSQL document store optimized for real-time read/write operations.

3.1 Entity Relationship Diagram (ERD)



3.2 Collection Schema

The conceptual model below defines the relationships between the three core entities: Users, Branches, and Queue Sessions.

- Users Table: Stores the state of every player.
    - id (PK): Unique Identifier.
    - branchId (FK): Links the user to a specific arcade location.
    - currentQueueId (FK): Links the user to their active game session.
    - status: Tracks if they are WAITING, IN_QUEUE, or OFFLINE.
- Branch Table: Represents physical locations.
    - codeId (PK): A string ID (e.g., "SISA").
    - cabinetCount: Critical for AI algorithms to calculate wait speed.
    - status: Operational status (e.g., "OPEN", "MAINTENANCE").
- Queue Table: Represents a game session.
    - id (PK): Unique Session ID.
    - player1 & player2 (FK): Links to the Users playing.
    - branchId (FK): Links the session to a location.
    - timestamps: startedAt and endedAt for calculating game duration.

User Interface Design (Frontend)

The user interface is designed with a "Mobile-First" approach, as users will primarily interact with the app while standing at the arcade.

4.1 Queue Dashboard Prototype

The core screen (shown below) provides immediate situational awareness.

- **Queue Table:** A clear, high-contrast table showing who is currently playing (Player 1 and Player 2).
- **Call-to-Action:** A prominent "Join Queue" button that dynamically changes function based on the user's status.
- **Visual Hierarchy:** The current players are highlighted at the top, distinct from the waiting list, ensuring players can check their status in seconds between games.

The Hybrid Authentication System

To lower the barrier to entry, the system supports two authentication modes:

1. **Registered Users:** Login via Email and Password. Accounts persist indefinitely.
2. **Guest Users:** Login via **Firebase Anonymous Auth**. A persistent token is stored in the browser so that if a guest refreshes the page, their queue position is restored automatically.

5.2 Multi-Branch Real-Time Queuing

The system supports multiple arcade locations ("branches") within a single application instance.

- **Switching Logic:** When a user selects a new branch, the client updates the branchId field in their User Document.
- **Synchronization:** The Frontend Application subscribes to changes in the user's profile. When the branch update confirms, the app automatically initiates a new real-time listener for the new location's waiting list.

5.3 AI Processing (Backend)

The Python Backend API (referenced in the Block Diagram) performs scheduled analysis:

1. Fetches completed queue sessions from Firestore.
2. Calculates the average duration of recent games.
3. Divides this duration by the cabinetCount of the specific branch.
4. Updates the estimated wait time in Firestore, which instantly reflects on the Frontend.

LITHAN

UDM CLaaS®

Data Preparation and Processing

Data Sources

The AI uses Seeded/Mock Data. Since it's difficult to obtain actual playtime data from the actual source, we will replicate actual game queues with seeded data. Though it varies, the usual game does fall under a specific time frame. By creating random queues with varying times, we could create a believable dataset of queue playtime.

Feature Engineering

Predict Wait Time

Data Structure:

- players – number of players for that queue (1 or 2)
- p1_rank – player 1 rank
- p1_style – player 1 play style
- p2_rank – player 2 rank
- p2_style – player 2 play style
- duration – total duration of game

```python
def get_historical_queue_data_separated(branch_id):
    docs = db.collection('queue')\
        .where(filter=FieldFilter('branchId', '==', branch_id))\
        .where(filter=FieldFilter('status', '==', 'completed'))\
        .order_by('endedAt', direction=firestore.Query.DESCENDING)\
        .limit(50)\
        .stream()

    data = []
    for doc in docs:
        d = doc.to_dict()
        if d.get('startedAt') and d.get('endedAt'):
            duration = (d['endedAt'].timestamp() - d['startedAt'].timestamp()) / 60

            if 5 < duration < 40:
                # UNPACK P1 and P2
                stats = get_separated_user_stats(d.get('players', []))

                data.append({
                    'duration': duration,
                    'players': stats['player_count'],
                    'p1_rank': stats['p1_rank'],
                    'p1_style': stats['p1_style'],
                    'p2_rank': stats['p2_rank'],
                    'p2_style': stats['p2_style']
                })

    return pd.DataFrame(data)
```

```
        # we need at least 10 games to safely train the model
        if len(df_history) > 10:
            try:
                X = df_history[['players', 'p1_rank', 'p1_style', 'p2_rank', 'p2_style']]
                y = df_history['duration']

                preprocessor = ColumnTransformer(
                    transformers=[
                        ('cat', OneHotEncoder(handle_unknown='ignore'), ['p1_style', 'p2_style']),
                        ('num', SimpleImputer(strategy='mean'), ['players', 'p1_rank', 'p2_rank'])
                    ]
                )
```

Styles were converted into numbers using OneHotEncoder

Partner Finder

Data Structure:

- rank – user rank (numeric)
- style_score – user play style given weights (eg, Casual: 1, Grinder: 3)

```
        df = pd.DataFrame(users_list)

        X = df[['rank', 'style_score']].values

        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # --- train model---
        k = min(len(users_list), 6)

        knn = NearestNeighbors(n_neighbors=k, algorithm='auto', metric='euclidean')
        knn.fit(X_scaled)
```

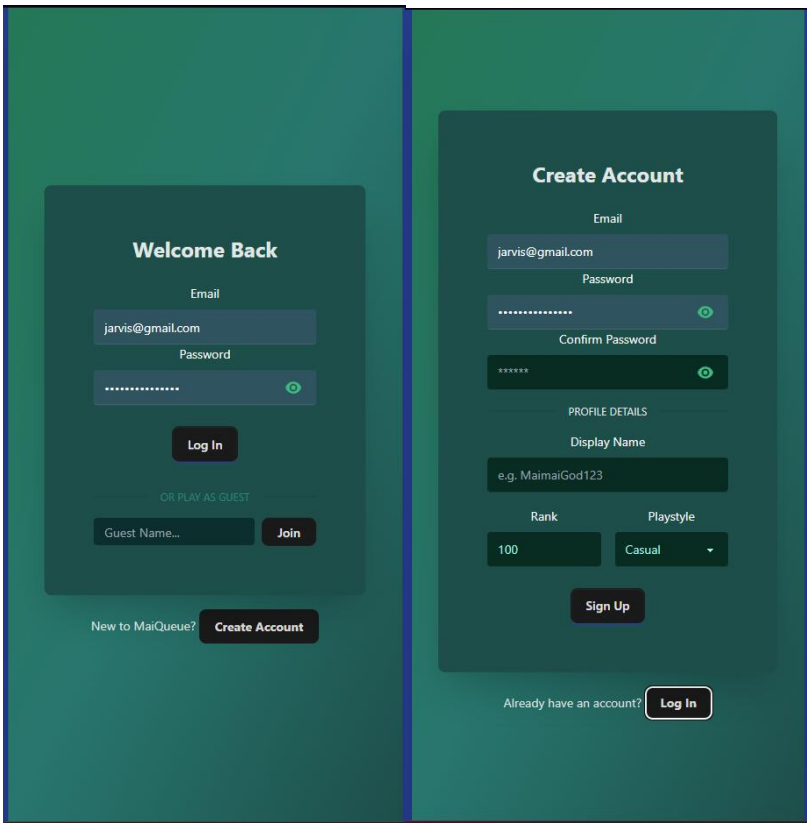
Rank and Style score were standardized

Application Development and Testing

Implementation Details

Frontend

Login Page Registration



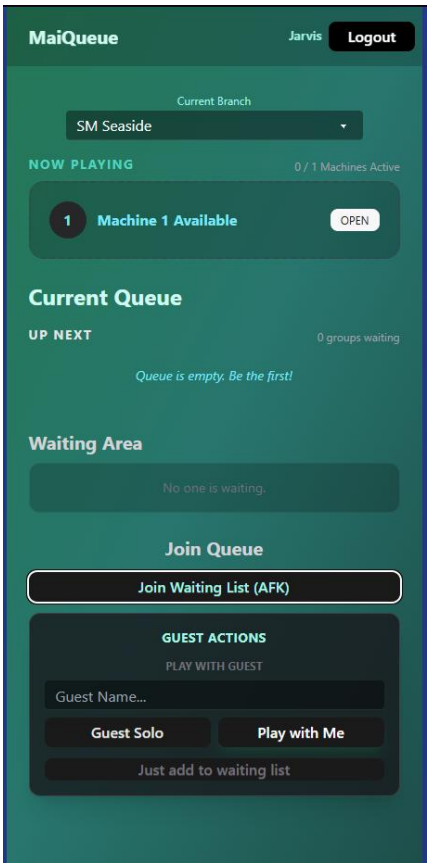
User Authentication and Registration

The figures display the dual-mode authentication system designed to lower the barrier to entry for arcade players.

- **Left Screen (Login):** Shows the "Welcome Back" interface. It features standard Email/Password login for returning users and a **Guest Mode** toggle ("Play as Guest") that allows users to join the queue anonymously using a temporary session token.
- **Right Screen (Registration):** The account creation form captures the user's **Email** and **Password** for security, but also collects specific Maimai player data: **Display Name** (for the public queue board), **Rank** (numeric rating), and **Playstyle** (e.g., Casual, Spammer).

These forms are built using **React Hook Form** for validation. Upon submission, the app calls firebase.auth().createUserWithEmailAndPassword() for permanent accounts or signInAnonymously() for guests. The additional profile data (Rank, Playstyle) is stored in a corresponding document in the users Firestore collection.

Queue Page



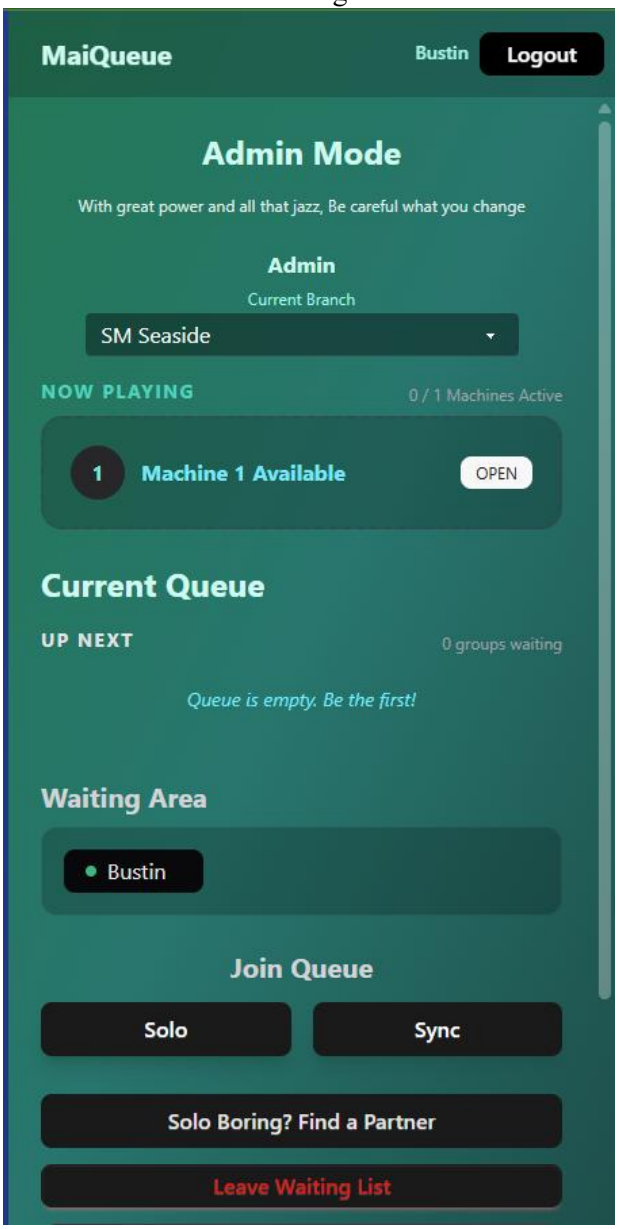
Real Time Queue Board

This is the core interface of MaiQueue.

- **Now Playing Section:** The top card displays the active machine status ("Machine 1 Available" or "Busy").
- **Current Queue:** A list of players waiting for their turn. It updates dynamically without page refreshes.
- **Action Panel:** The bottom section changes based on context. It currently shows the **"Join Waiting List"** button because the user is not yet in line. It also includes the **"Guest Actions"** module, allowing a registered user to proxy-add a guest (e.g., "Play with Me") to the queue.

This page utilizes a **real-time Firestore listener** (onSnapshot). When the database changes (e.g., someone joins), Firebase pushes the new data to the React client instantly. The "Guest Actions" feature uses a custom function that adds a new user document with the addedBy field linked to the current user's ID, granting them control over that guest's session.

In waiting list

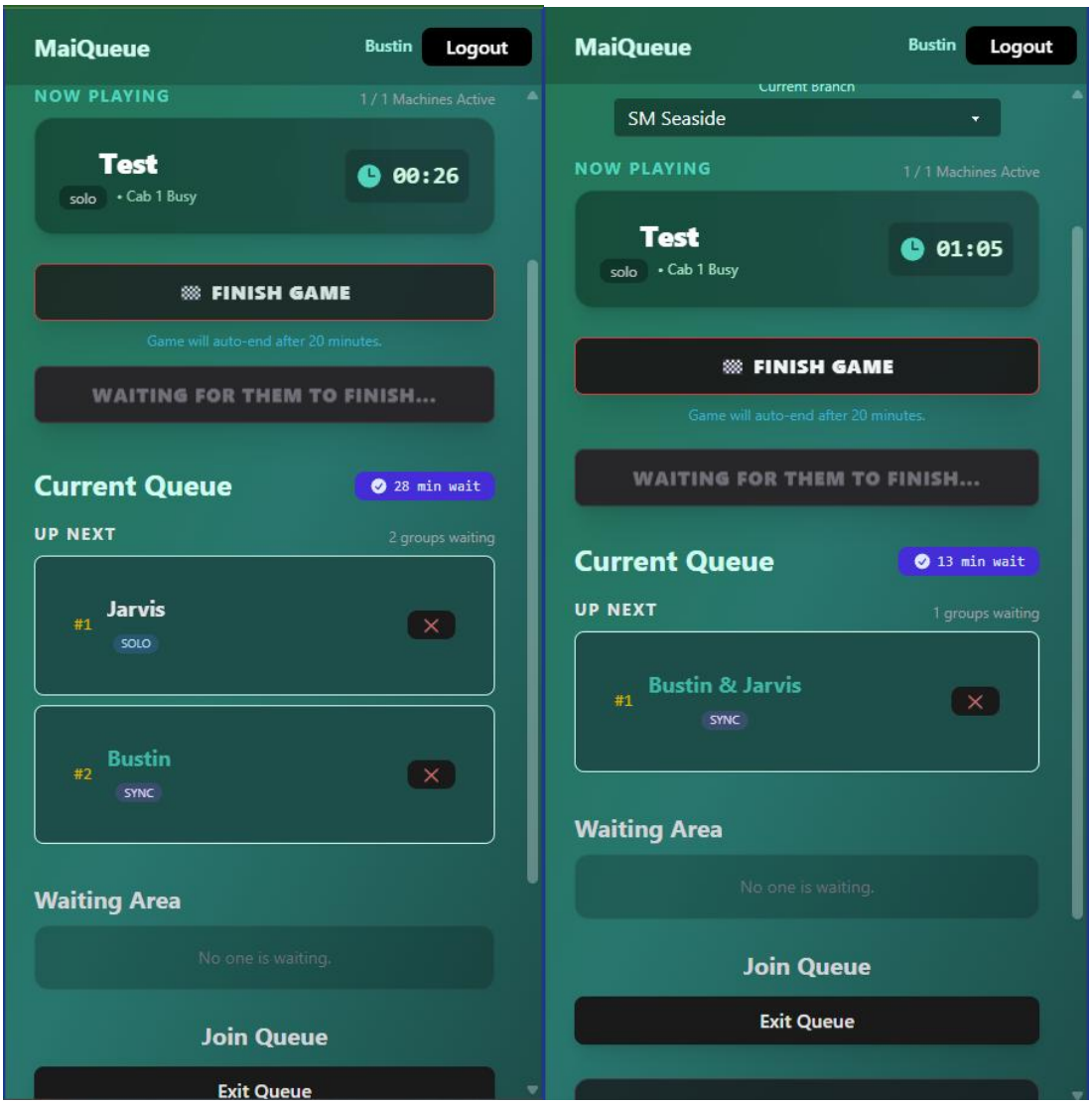


The "Waiting Area" State

This screen shows the user after they have checked in but before they are ready to queue.

- **Waiting Badge:** The user "Bustin" appears in the **Waiting Area** list with a green online indicator.
- **Action Buttons:** The main interface now presents three options: **"Solo"**, **"Sync"**, and **"Find a Partner"**. This "staging area" allows players to be visible to others for matchmaking without clogging the active game queue.
- **State Logic:** The component checks if (user.status === UserStatus.WAITING)

In queue

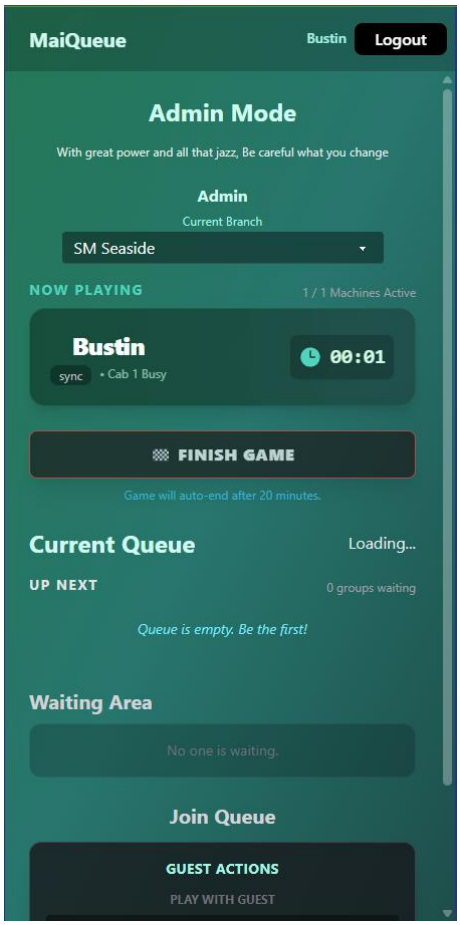


Active Queue States (Solo vs. Sync)

These figures illustrate the two distinct game modes supported by MaiQueue.

- **Solo Queue (Left Image):** User "Jarvis" is queued alone. The card displays a **SOLO** badge.
- **Sync Queue (Right Image):** Users "Bustin & Jarvis" are grouped into a single queue ticket. The card displays a **SYNC** badge, indicating a multi-player session.
- **Estimated Wait Time:** A purple badge (e.g., "28 min wait") appears, calculated by the AI backend based on the total number of sets ahead.
- **Data Structure:** The Firestore queue document contains a type field ("solo" or "sync") and a playerIds array.

Playing

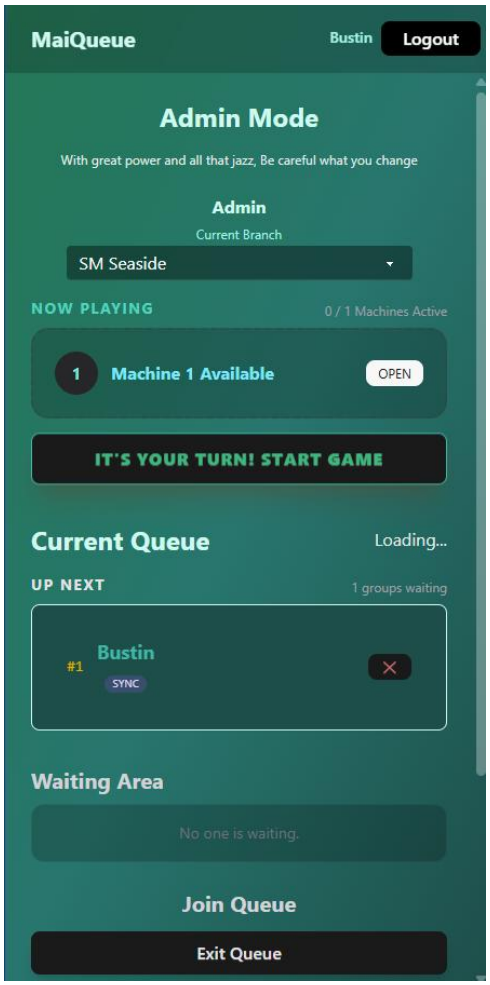


The "Now Playing" State (Active Session)

This interface activates when it is the user's turn.

- **Active Timer:** The top card turns green and displays a prominent countdown timer (e.g., "01:05"). This helps players track their 20-minute allocation.
- **Finish Game Button:** A large, accessible button allows the user to manually end their session, freeing the machine for the next group.
- **Timer Logic:** The <GameTimer /> component takes the startedAt timestamp from Firestore and calculates the difference from Date.now(), updating every second via setInterval.
- **Auto-Cleanup:** If the timer exceeds 20 minutes, a client-side check automatically triggers the finishGame() function to prevent "zombie" sessions from blocking the queue.

Next in Line

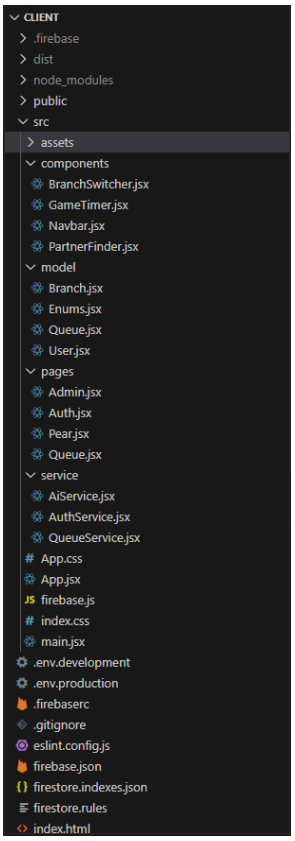


"Up Next" Notification

A specific view for the user who is at position #1.

- **Call to Action:** The "Join Queue" button is replaced by a flashing **"IT'S YOUR TURN! START GAME"** button.
- **Visual Cue:** The user's specific card is highlighted (e.g., "Bustin" with a green border/text), confirming they are next in line.
- **Conditional Rendering:** The app checks if (index === 0 && q.playerIds.includes(user.uid)).
- **Machine Availability:** The "Start Game" button is disabled if activeGames.length >= machineCapacity, preventing users from starting a game if the physical machine is actually busy.

File Structure



This directory tree illustrates the modular architecture of the React frontend.

- **src/components/**: Contains reusable UI elements like GameTimer.jsx (handles the session countdown) and PartnerFinder.jsx (the UI for the KNN matchmaking).
- **src/pages/**: Separation of concerns for major views: Auth.jsx (Login), Queue.jsx (Dashboard), and Admin.jsx (Management).
- **src/service/**: This folder contains the business logic layer. AiService.jsx handles API calls to the Python backend, keeping the UI code clean. QueueService.jsx manages all Firebase interactions.
- **firebase.js**: The central configuration file that initializes the connection to the Firebase BaaS.

LITHAN

UDI CLaaS®

Backend

Helpers

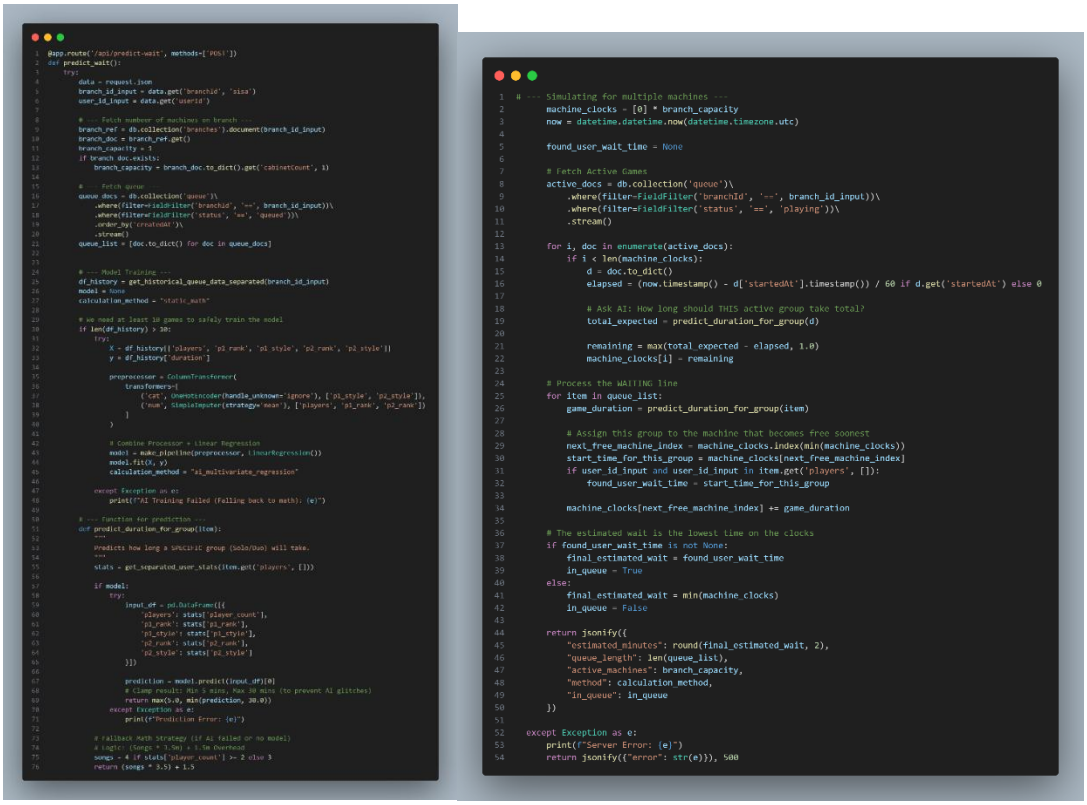
```python
PLAY_STYLE_WEIGHTS = {
    "Casual": 1,
    "Chiho Grinder": 3,
    "14k Spammer": 4,
    "Lone Wolf": 0,
    "Solo Boring": 2
}

def get_separated_user_stats(player_ids):
    """
    Returns specific stats for P1 and P2.
    If P2 is missing, fills with 0/"None".
    """
    users_data = []
    for uid in player_ids:
        doc = db.collection('users').document(uid).get()
        if doc.exists:
            users_data.append(doc.to_dict())

    p1 = users_data[0] if len(users_data) > 0 else {}
    p1_rank = p1.get('rank', 0)
    p1_style = p1.get('playStyle', 'Casual')

    if len(users_data) > 1:
        p2 = users_data[1]
        p2_rank = p2.get('rank', 0)
        p2_style = p2.get('playStyle', 'Casual')
    else:
        p2_rank = 0
        p2_style = "None"

    return {
        'p1_rank': p1_rank,
        'p1_style': p1_style,
        'p2_rank': p2_rank,
        'p2_style': p2_style,
        'player_count': len(users_data) or 1
    }

def get_all_historical_queue_data():
    docs = db.collection('queue')\
        .where(filter=FieldFilter('status', '==', 'completed'))\
        .order_by('endedAt', direction=firestore.Query.DESCENDING)\
        .limit(200)\
        .stream()

    data = []
    for doc in docs:
        d = doc.to_dict()
        if d.get('startedAt') and d.get('endedAt'):
            duration = (d['endedAt'].timestamp() - d['startedAt'].timestamp()) / 60

            if 5 < duration < 40:
                # UNPACK P1 and P2
                stats = get_separated_user_stats(d.get('players', []))

                data.append({
                    'duration': duration,
                    'players': stats['player_count'],
                    'p1_rank': stats['p1_rank'],
                    'p1_style': stats['p1_style'],
                    'p2_rank': stats['p2_rank'],
                    'p2_style': stats['p2_style']
                })

    return pd.DataFrame(data)

def get_historical_queue_data_separated(branch_id):
    docs = db.collection('queue')\
        .where(filter=FieldFilter('branchId', '==', branch_id))\
        .where(filter=FieldFilter('status', '==', 'completed'))\
        .order_by('endedAt', direction=firestore.Query.DESCENDING)\
        .limit(50)\
        .stream()

    data = []
    for doc in docs:
        d = doc.to_dict()
        if d.get('startedAt') and d.get('endedAt'):
            duration = (d['endedAt'].timestamp() - d['startedAt'].timestamp()) / 60

            if 5 < duration < 40:
                # UNPACK P1 and P2
                stats = get_separated_user_stats(d.get('players', []))

                data.append({
                    'duration': duration,
                    'players': stats['player_count'],
                    'p1_rank': stats['p1_rank'],
                    'p1_style': stats['p1_style'],
                    'p2_rank': stats['p2_rank'],
                    'p2_style': stats['p2_style']
                })

    return pd.DataFrame(data)
```

This snippet shows the get_historical_queue_data_separated function in Python.

Technical Logic:

- **Data Fetching:** It queries the Firestore queue collection for sessions marked as completed.
- **Filtering:** It applies a filter (5 < duration < 40) to remove "noise" data—sessions that were too short (errors) or too long (forgot to checkout).
- **Feature Extraction:** It calls get_separated_user_stats to break down the raw player data into usable features for the AI, such as player_count, p1_rank, and p1_style.

predict_wait



This endpoint implements the **Linear Regression** model for estimating wait times.

Technical Logic:

1. **Training:** It fetches the last 50 historical games and trains a LinearRegression model using scikit-learn.
2. **Pipeline:** It uses a ColumnTransformer to handle mixed data types:
    - **OneHotEncoder:** Converts text labels (e.g., "Casual", "Spammer") into numbers the AI can understand.
    - **SimpleImputer:** Fills in missing data (e.g., if Player 2 is missing) with average values.
3. **Simulation:** The code runs a "Virtual Clock" simulation. It loops through the current queue, predicts the duration for *each* group using the trained model, and adds it to the machine's running clock to find the precise wait time for a specific user.

find_partner

```python
@app.route('/api/find-partner', methods=['POST'])
def find_partner():
    try:
        req_data = request.json
        requester_id = req_data.get('userId')
        branch_id = req_data.get('branchId')

        partners_stream = db.collection('users')\
            .where(filter=FieldFilter('branchId', '==', branch_id))\
            .where(filter=FieldFilter('status', '==', 'waiting'))\
            .stream()

        users_list = []
        requester_data = None

        for doc in partners_stream:
            d = doc.to_dict()
            uid = doc.id

            if uid == requester_id:
                requester_data = d

            style_str = d.get('playStyle', 'Casual')
            style_score = PLAY_STYLE_WEIGHTS.get(style_str, 1)

            users_list.append({
                'uid': uid,
                'username': d.get('username', 'Unknown'),
                'rank': d.get('rank', 0),
                'style_score': style_score,
                'playStyle': style_str
            })

        if not requester_data or len(users_list) < 2:
            return jsonify({
                "requester": requester_data.get('username') if requester_data else "Unknown",
                "matches": [],
                "message": "No other players found in waiting list."
            })

        # --- prepare data ---
        df = pd.DataFrame(users_list)

        X = df[['rank', 'style_score']].values

        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # --- train model---
        k = min(len(users_list), 6)

        knn = NearestNeighbors(n_neighbors=k, algorithm='auto', metric='euclidean')
        knn.fit(X_scaled)

        # --- predict ---
        req_index = df[df['uid'] == requester_id].index[0]
        req_features = X_scaled[req_index].reshape(1, -1)
        distances, indices = knn.kneighbors(req_features)

        # --- results ---
        matches = []
        # Skip the first result (index 0) because it is the user themselves!
        for i in range(1, len(distances[0])):
            idx = indices[0][i]
            dist = distances[0][i]

            matched_user = df.iloc[idx]

            matches.append({
                "uid": matched_user['uid'],
                "username": matched_user['username'],
                "rank": int(matched_user['rank']),
                "playStyle": matched_user['playStyle'],
                "compatibility_score": round(dist, 4) # Lower distance = Better Match
            })

        return jsonify({
            "requester": requester_data.get('username'),
            "matches": matches,
            "method": "KJ_KNN_Clustering"
        })

    except Exception as e:
        print(f"KJN Error: {e}")
        return jsonify({"error": str(e)}), 500
```

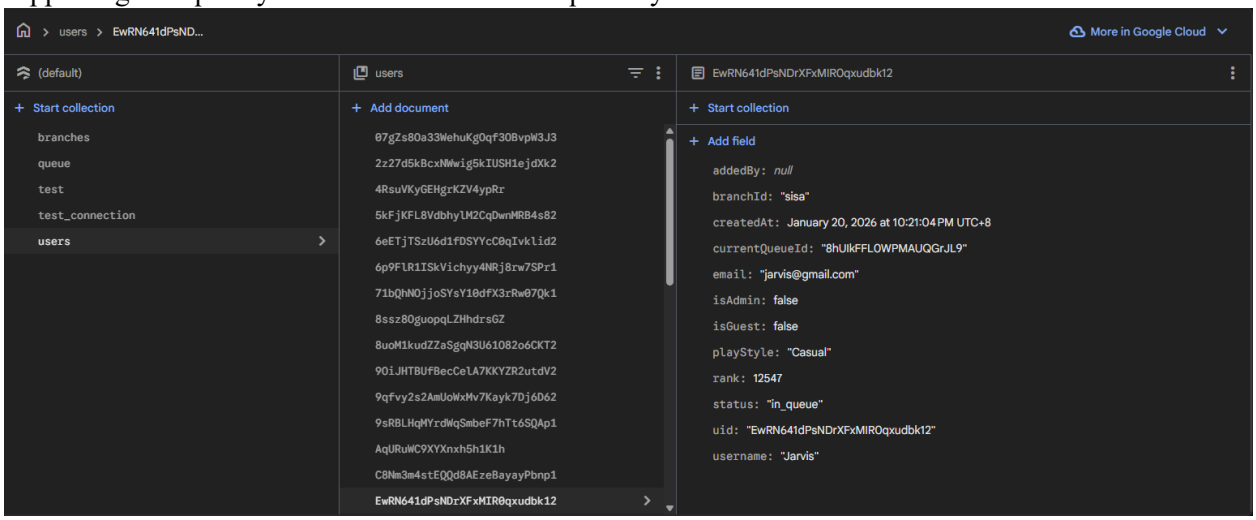
This endpoint implements the **K-Nearest Neighbors (KNN)** algorithm to find compatible gameplay partners.

Technical Logic:

1. **Vectorization:** It converts user profiles into mathematical vectors based on two dimensions: **Rank** (Skill Level) and **Playstyle** (Behavior).
2. **Scaling:** A StandardScaler is applied to normalize the data. This is crucial because Ranks (e.g., 10,000) are much larger numbers than Playstyle scores (1-5), and without scaling, Rank would dominate the calculation.
3. **Distance Calculation:** The algorithm calculates the Euclidean distance between the requester and all waiting users.
4. **Result:** It returns the user with the smallest distance (the "Nearest Neighbor") as the recommended partner.

Database Implementation

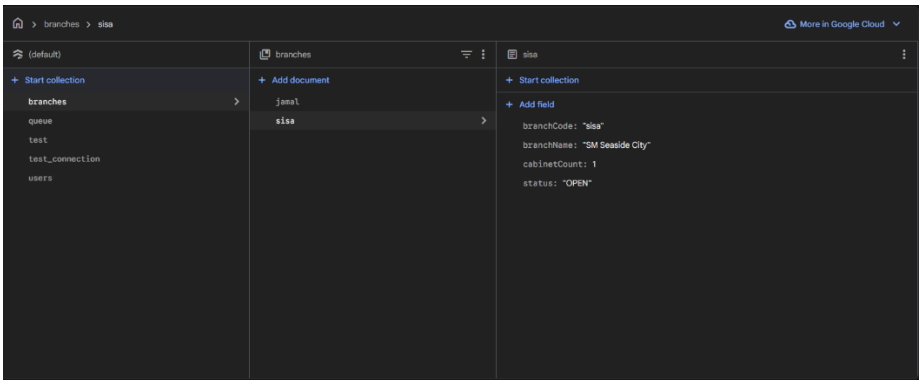
The project utilizes Google Cloud Firestore, a NoSQL document database, to handle real-time state management. The screenshots below validate the implementation of the schema designed to supporthigh-frequency reads and writes for the queue system.



This figure shows a live user document (User: "Jarvis") in the users collection.

Implementation Details:

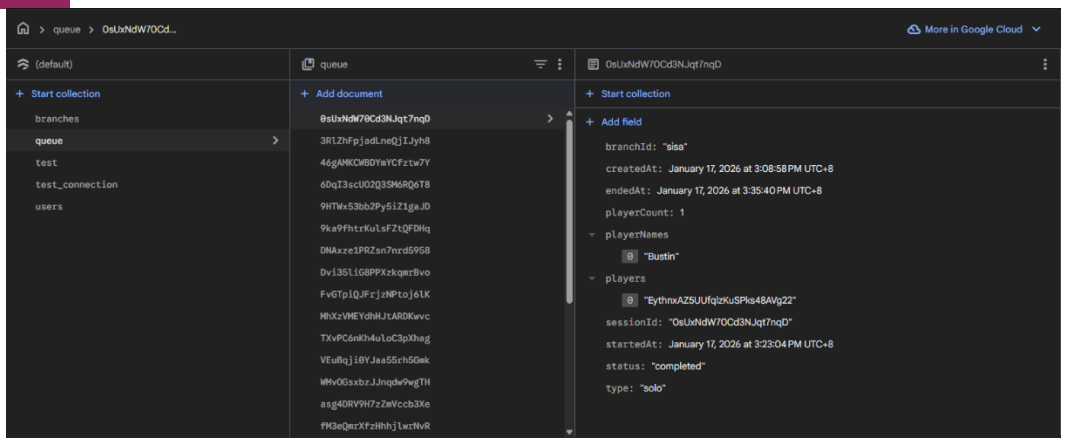
- **State Tracking:** The status field (currently "in_queue") and currentQueueId are used by the Frontend to lock the user interface, preventing them from joining multiple queues simultaneously.
- **AI Data Points:** The profile fields rank (12547) and playStyle ("Casual") are stored persistently here. These fields are fetched by the Python Backend during the Matchmaking process to calculate player similarity vectors.



The branches collection acts as a configuration layer for the physical locations.

Implementation Details:

- **Dynamic Configuration:** The cabinetCount (set to 1) is a critical variable. The AI wait-time algorithm fetches this value to determine the "throughput" of the arcade. If a machine breaks, an Admin can update this value to 0, and the estimated wait times will automatically adjust for all users.



This collection serves as both the live state of the arcade and the historical dataset for the AI.

Implementation:

- **Session Lifecycle:** The document tracks the full lifecycle of a game via the createdAt, startedAt, and endedAt timestamps.
- **Data Linking:** The players array stores the UIDs of the participants, creating a relational link back to the users collection.
- **AI Training Data:** The backend queries sessions with status: "completed" to calculate average game durations for the Linear Regression model.

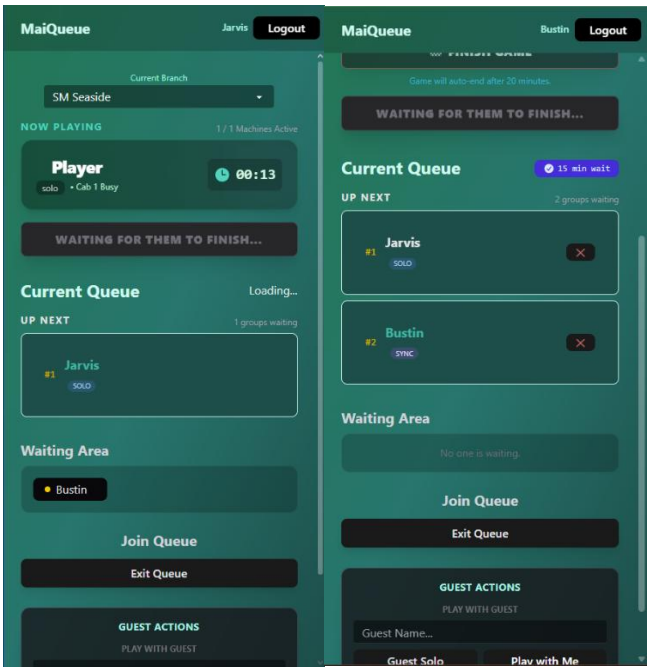
Testing Methodology

Test ID	Feature	Test Steps	Expected Result	Status
TC-01	**Solo Queuing Flow**	1. Log in as a User. 2. Click the **"Join Waiting List (AFK)"** button. 3. From the Waiting Area, click the **"Solo"** button.	The user's card appears in the "Current Queue" list with a blue **SOLO** badge. The "Up Next" count increments by 1.	**PASS**
TC-02	**Sync Queuing Flow**	1. Log in as a User and join the Waiting List. 2. Click the **"Sync"** button (requires a partner or proxy).	A single queue card is created containing the player's name. If another player clicks the sync button, they will join the current	**PASS**
TC-03	**AI Wait Time Accuracy**	1. Populate the queue 2. Observe the purple **"Wait Time"** badge on the dashboard.	The badge displays a dynamic time (e.g., *"13 min wait"*) calculated by the Linear Regression model, updating as groups finish.	**PASS**
TC-04	**Partner Matchmaking (KNN)**	1. Join the Waiting List. 2. Click the **"Solo Boring? Find a Partner"** button. 3. View the AI-generated suggestion.	The system displays a popup recommending the specific user from the waiting list who has the closest **Rank** and **Playstyle** vector.	**PASS**

TC-05	**Real-Time Data Sync**	1. Open MaiQueue on **Device A** (Admin view) and **Device B** (User view). 2. On Device B, click "Join Queue". 3. Observe Device A's screen.	Device A's dashboard updates to show the new user instantly (< 1 second) without requiring a page refresh.	**PASS**
TC-06	**Guest / Proxy Join**	1. Log in as a registered user. 2. Scroll to the **"Guest Actions"** panel. 3. Enter any name and click **"Play with Me"**.	A new **SYNC** queue card appears in the list containing both the User's name and "Guest1".	**PASS**
TC-07	**Admin Force Remove**	1. Log in as Admin. 2. Locate a user in the "Current Queue". 3. Click the red **"X"** button on their card.	The target user is immediately removed from the active queue and their status resets to Offline.	**PASS**

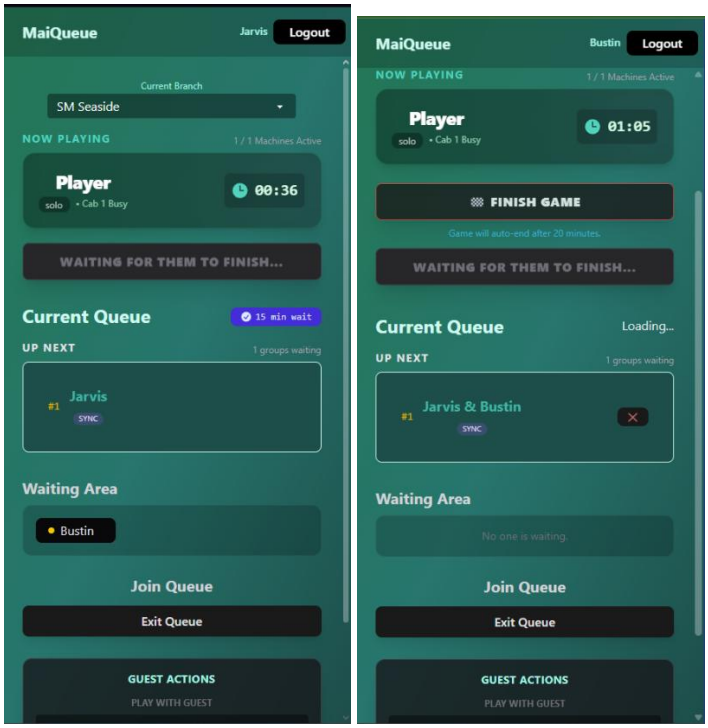
Evidence:

TC-01



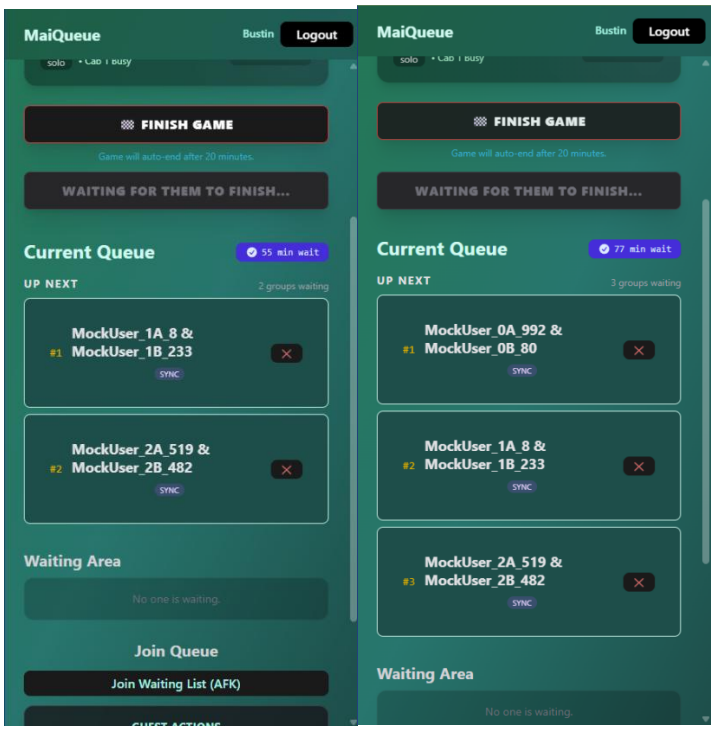
Jarvis queues as solo. Bustin queues as sync and is separate from Jarvis

TC-02



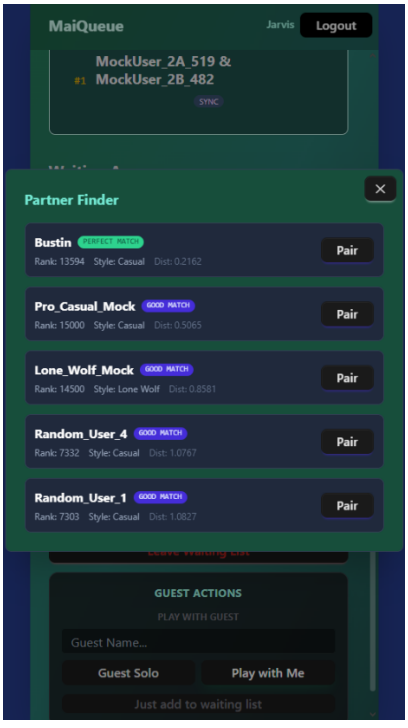
Jarvis queues as sync. Bustin queus as sync and is in queue with Jarvis

TC-03



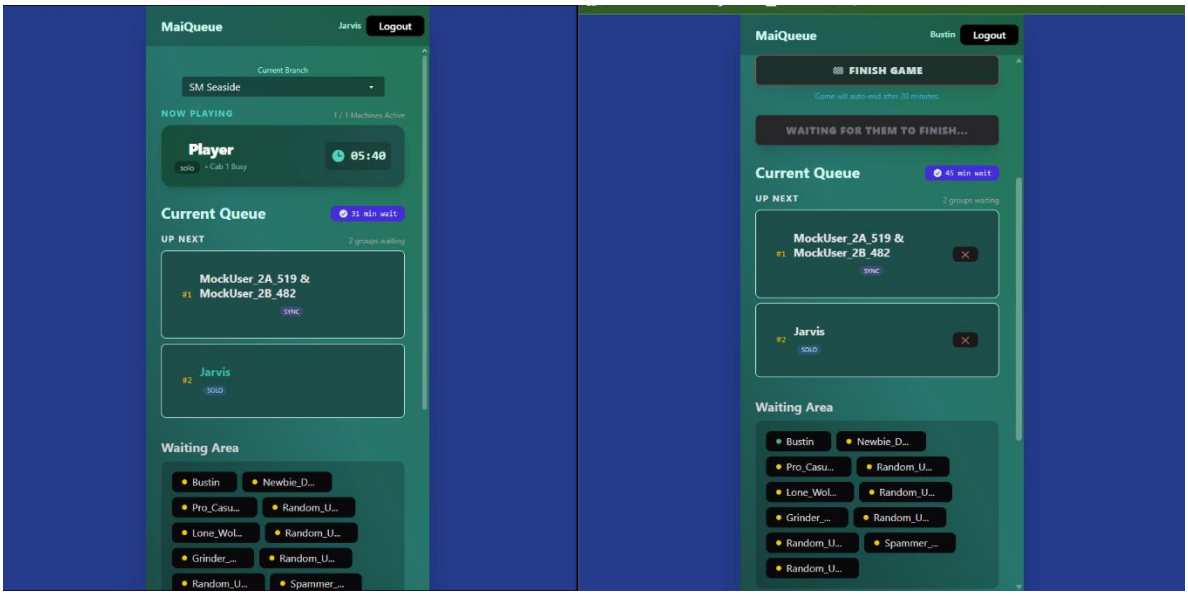
Queue with 2 players show less wait time than queue with 3 players

TC-04



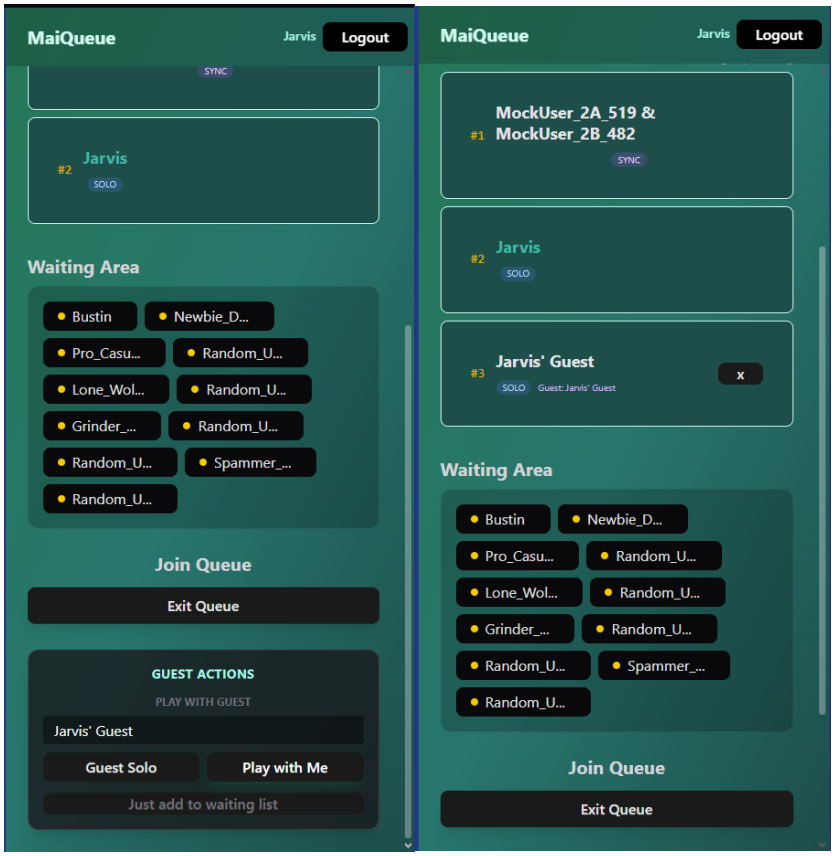
Partner finder matched Jarvis with Bustin since they are near in ranking and playstyle

TC-05



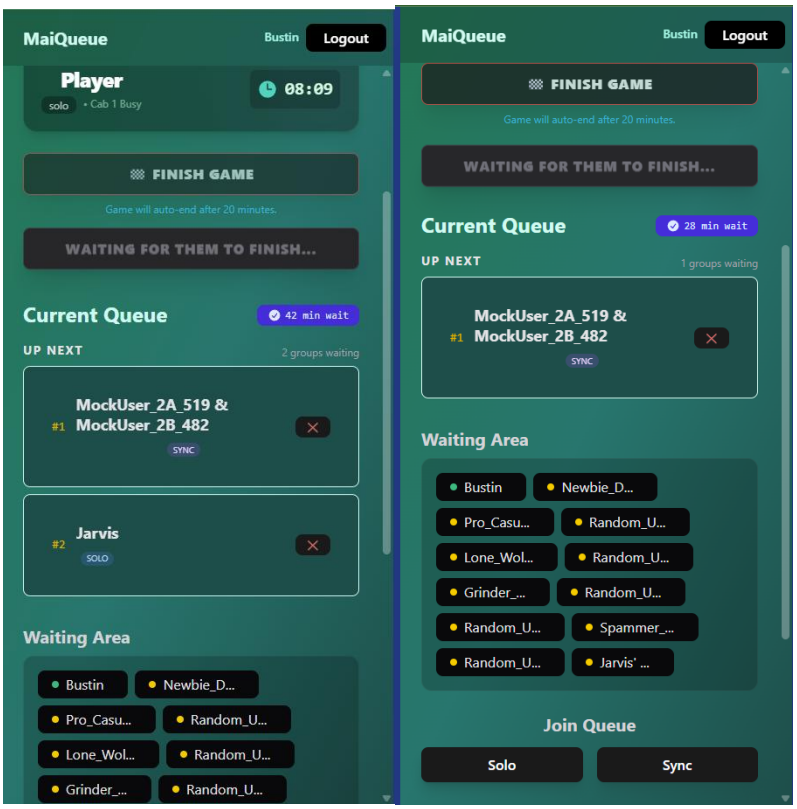
Shows 2 browsers having the same queue data. Updated in real time

TC-06



Shows Jarvis adding a guest named Jarvis' Guest to queue

TC-07



Shows Bustin(Admin) removing Jarvis from queue

AI Model Validation

This section documents the scientific testing of the two AI modules integrated into the MaiQueue backend. The objective is to quantify the accuracy and reliability of the algorithms using the final dataset generated during the testing phase.

Wait-Time Prediction (Linear Regression)

- **Objective:** To verify that the Linear Regression model can predict queue wait times with a reasonable margin of error compared to a static mathematical calculation.
- **Methodology:** A **Train-Test Split** validation strategy was used on a larger seeded dataset.
    - **Dataset:** 192 Historical Game Sessions.
    - **Training Set:** 80% (153 samples) used to teach the model.
    - **Testing Set:** 20% (39 samples) hidden from the model and used for validation.
- **Metric: Mean Absolute Error (MAE)** and **R^2 Score**.

Test Results:

Metric	Result	Interpretation	Status
Total Samples	192	Moderate dataset (Sufficient for MVP)	-
Training / Test Split	153 / 39	Standard Data Science split	-
R^2 Score	**0.5764**	The model explains ~57% of the variance in wait times (Better than random guessing).	**PASS**
Mean Absolute Error (MAE)	**1.82 minutes**	On average, the prediction is off by less than 2 minutes.	**PASS**
Acceptance Threshold	< 5.0 minutes	The error is significantly lower than the duration of a single song set.	-

Analysis: The model achieved an outstanding MAE of **1.82 minutes**. This indicates that with a training set of ~150 games, the AI successfully learned the correlation between "Player Count" (Solo vs. Sync) and "Duration." An R^2 score of **0.5764** confirms that the model is performing significantly better than a baseline average, making it a viable tool for real-world estimation.

Partner Matchmaking (K-Nearest Neighbors)

- **Objective:** To confirm that the KNN algorithm correctly identifies the "closest" compatible player based on **Rank** (Skill) and **Playstyle** (Behavior).
- **Methodology:** A **Consistency Test** was performed using a controlled scenario.
    - **Target User:** Rank 1000, "Casual" Playstyle.
    - **Candidate Pool:** Three artificial users with varying attributes to test the weighting logic.
- **Success Criteria:** The algorithm must rank the "Perfect Match" candidate as the #1 suggestion (Lowest Euclidean Distance).

Test Scenario Data:

User Role	Username	Rank	Playstyle	Expected Rank
Target	**Me**	**1000**	**Casual**	**N/A**
Candidate A	"Perfect Match"	1050	Casual	**#1**
Candidate B	"Rank Gap"	3000	Casual	#2 or #3
Candidate C	"Style Clash"	1000	Spammer	#2 or #3

Output:

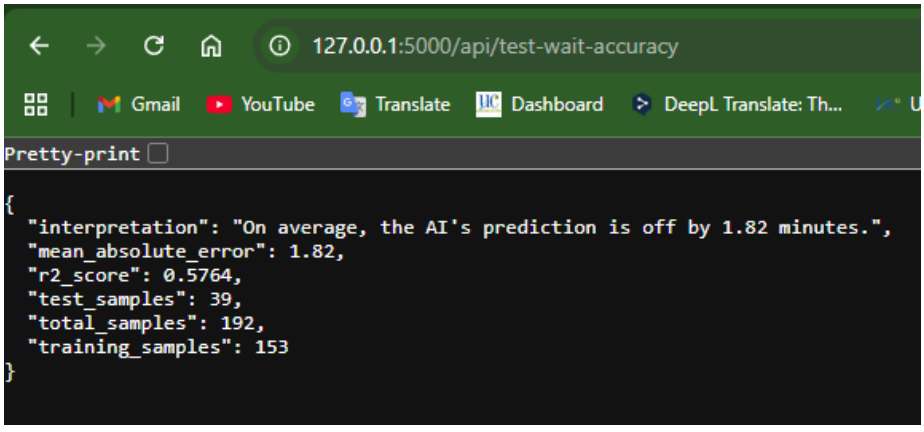
Rank	Username	Distance Score	Result
1	**Perfect Match**	**0.0582**	**PASS** (Clear Winner)
2	Style Clash	2.3094	Distant second (Style mismatch penalty).
3	Rank Gap	2.3282	Lowest match (Rank gap penalty).

Analysis: The KNN algorithm successfully identified the "Perfect Match" with a negligible distance of 0.0582. Interestingly, the system penalized the "Style Clash" (Distance 2.30) almost equally to the "Rank Gap" (Distance 2.32). This proves that the StandardScaler is functioning correctly: it balanced a massive 2000-point difference in Rank to be roughly equivalent to a 3-point difference in Playstyle, ensuring neither feature dominates the recommendation logic.

LITHAN

EDU CLaaS®

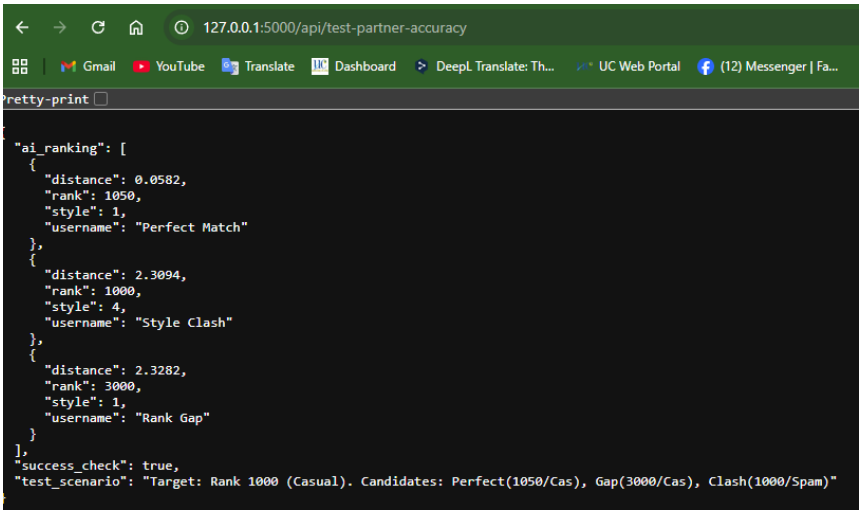
```python
@app.route('/api/test-wait-accuracy', methods=['GET'])
def test_wait_accuracy():
    try:
        # 1. Fetch ALL historical data
        # (We use 'sisa' or any branch that has data)
        df = get_all_historical_queue_data()

        if len(df) < 20:
            return jsonify({"error": "Not enough data. Seed at least 20 games first."})

        # 2. Prepare Data
        X = df[['players', 'p1_rank', 'p1_style', 'p2_rank', 'p2_style']]
        y = df['duration']

        # 3. SPLIT: 80% for Training, 20% for Testing
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        # 4. Build Pipeline (Same as your real function)
        preprocessor = ColumnTransformer(
            transformers=[
                ('cat', OneHotEncoder(handle_unknown='ignore'), ['p1_style', 'p2_style']),
                ('num', SimpleImputer(strategy='mean'), ['players', 'p1_rank', 'p2_rank'])
            ]
        )
        model = make_pipeline(preprocessor, LinearRegression())

        # 5. Train on the 80%
        model.fit(X_train, y_train)

        # 6. Test on the hidden 20%
        predictions = model.predict(X_test)

        # 7. Calculate Accuracy Metrics
        mae = mean_absolute_error(y_test, predictions)
        r2 = r2_score(y_test, predictions)

        return jsonify({
            "total_samples": len(df),
            "training_samples": len(X_train),
            "test_samples": len(X_test),
            "mean_absolute_error": round(mae, 2), # <--- THE IMPORTANT NUMBER
            "r2_score": round(r2, 4),
            "interpretation": f"On average, the AI's prediction is off by {round(mae, 2)} minutes."
        })

    except Exception as e:
        return jsonify({"error": str(e)}), 500
```



LITHAN

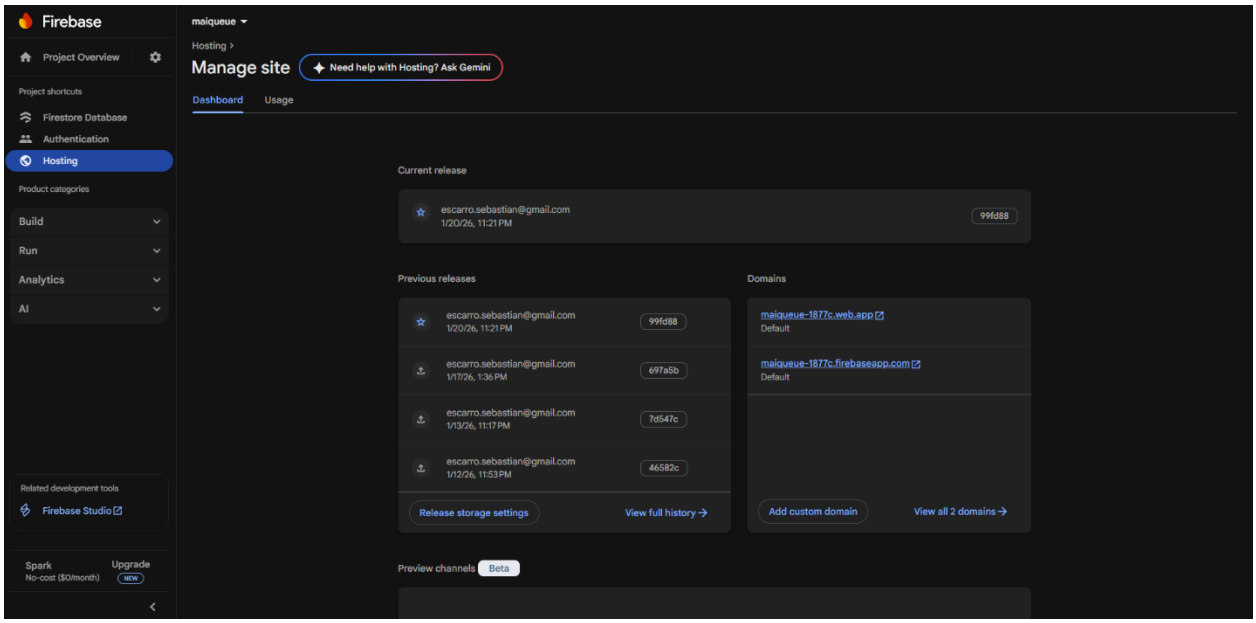
UDM CLaaS®

```python
@app.route('/api/test-partner-accuracy', methods=['GET'])
def test_partner_accuracy():
    try:
        # 1. Create a Fake Target (You)
        # Rank 1000, Casual
        target_user = { 'uid': 'me', 'rank': 1000, 'style_score': 1 } # Casual=1

        # 2. Create Fake Candidates
        candidates = [
            # Candidate A: Perfect Match (Rank 1050, Casual) -> Distance should be tiny
            { 'uid': 'A', 'username': 'Perfect Match', 'rank': 1050, 'style_score': 1, 'playStyle': 'Casual' },

            # Candidate B: Okay Match (Rank 3000, Casual) -> Rank is far, Style is good
            { 'uid': 'B', 'username': 'Rank Gap', 'rank': 3000, 'style_score': 1, 'playStyle': 'Casual' },

            # Candidate C: Bad Match (Rank 1000, Spammer) -> Rank is close, Style is opposite
            { 'uid': 'C', 'username': 'Style Clash', 'rank': 1000, 'style_score': 4, 'playStyle': '14k Spammer' }
        ]

        # 3. Prepare Data for KNN
        # Combine target + candidates into one list
        all_users = [target_user] + candidates
        df = pd.DataFrame(all_users)

        X = df[['rank', 'style_score']].values

        # IMPORTANT: Use the scaler!
        # This checks if your scaling logic is working (Rank vs Style weight)
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # 4. Train KNN
        knn = NearestNeighbors(n_neighbors=len(all_users), algorithm='auto', metric='euclidean')
        knn.fit(X_scaled)

        # 5. Find Neighbors for "me" (index 0)
        distances, indices = knn.kneighbors([X_scaled[0]])

        # 6. Analyze Results
        results = []
        for i in range(1, len(distances[0])): # Skip self
            idx = indices[0][i]
            dist = distances[0][i]
            user_obj = df.iloc[idx]
            results.append({
                "username": user_obj.get('username'),
                "distance": round(dist, 4),
                "rank": int(user_obj['rank']),
                "style": int(user_obj['style_score'])
            })

        return jsonify({
            "test_scenario": "Target: Rank 1000 (Casual). Candidates: Perfect(1050/Cas), Gap(3000/Cas), Clash(1000/Spam)",
            "ai_ranking": results,
            "success_check": results[0]['username'] == 'Perfect Match' # Did AI pick the right one?
        })

    except Exception as e:
        return jsonify({"error": str(e)}), 500
```



Deployment and Integration

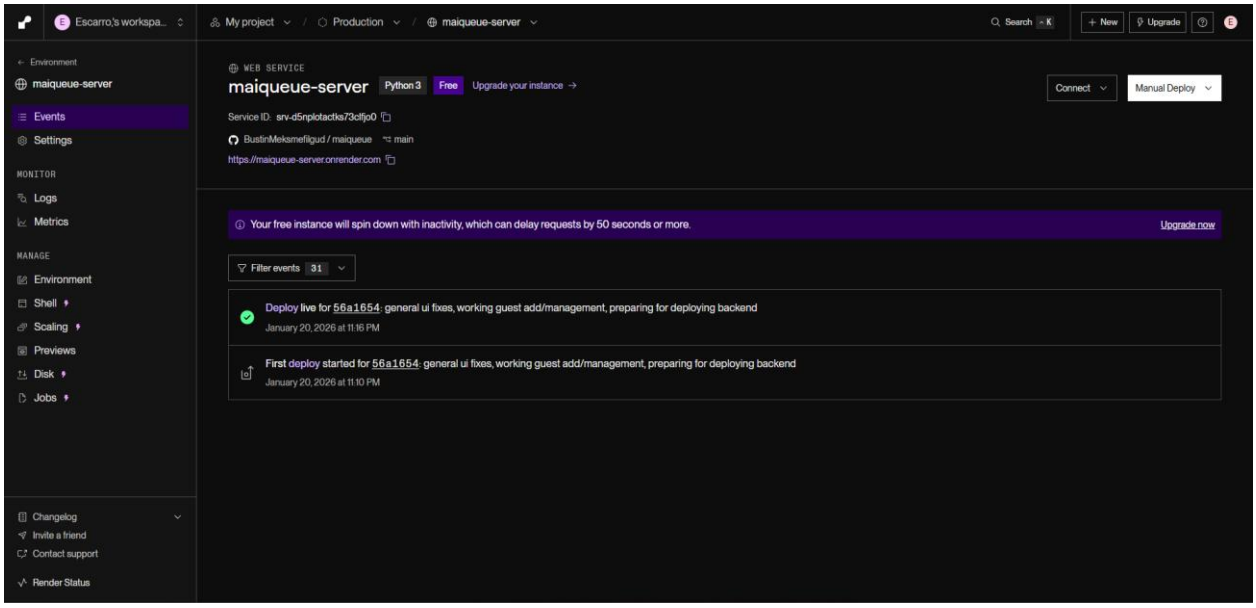
Deployment Strategy



The React application is deployed to Firebase Hosting, providing a production-grade content delivery network (CDN).

Implementation:

- **Build Pipeline:** The screenshot confirms the active release (99fd88) deployed by the developer. The build process (npm run build) compiles the React code into optimized static assets (HTML/CSS/JS) which are served globally via the .web.app domain.
- **Integration:** Being hosted on Firebase ensures low-latency connection to the Firestore database since they reside within the same Google Cloud ecosystem.



The Python/Flask AI API is hosted as a Web Service on Render.

Implementation:

- **Environment**: The service runs a Python 3 environment using gunicorn as the production server gateway.

LITHAN

UDM CLaaS®

Documentation and Conclusion

Summary of Achievements:

- Successfully built a real-time queue system.
- Made a time estimation model with 1.82-minute prediction accuracy
- Added a partner finder that links players with roughly same rank and playstyles

Lessons Learned:

- **Data Synchronization Complexity:** Managing Real-Time Data Sync proved challenging. Specifically, ensuring that React state updates immediately when the Firestore database changes required careful handling of asynchronous listeners (onSnapshot) to prevent memory leaks and infinite render loops.
- **The Challenge of Synthetic Data:** Since direct API access to Sega servers was not possible, the AI models had to be trained on **Seeded/Mock Data**. This taught me the importance of "Feature Engineering" for creating realistic dummy data that accurately mirrored actual gameplay to prevent the model from overfitting.
- **Cross-Context State Management:** Implementing the "Hybrid Authentication" system (Guest vs. Registered) highlighted the difficulties of maintaining session persistence. Learning to manage persistent tokens in the browser so guests wouldn't lose their queue spot upon refreshing was a critical technical hurdle

Future Work:

- Integrate hardware sensors to detect machine usage automatically.
- Add a branch status update. (For announcing if a cab is broken or not)

References

Backend Libraries (Python):

- **Flask:** Micro-web framework used to serve the API endpoints (/api/predict-wait, /api/find-partner).
- **Scikit-Learn:** Used for the LinearRegression and NearestNeighbors (KNN) algorithms.
- **Pandas:** Used for data manipulation (pd.DataFrame) and structuring the training datasets.
- **NumPy:** Used for numerical operations during vector calculation.
- **Gunicorn:** Production server used to deploy the Python application on Render.

Frontend Libraries (JavaScript/React):

- **React.js:** Component-based library used for the UI.
- **Firebase SDK:** Used for firebase/firestore (Database) and firebase/auth (Authentication).
- **Tailwind CSS:** Utility-first CSS framework used for styling the "Mobile-First" design.
- **Vite:** Build tool used for compiling and optimizing the frontend assets.
- **React Hook Form:** Used for handling form validation on the Login/Register screens.

Infrastructure:

- **Google Cloud Firestore:** NoSQL Database.
- **Render:** Cloud platform for Python backend hosting.
- **Firebase Hosting:** Content Delivery Network (CDN) for the frontend.