Lewis Head

Registration number 100375072

2025

Implementation of a Machine Learning Recommendation System

Supervised by Dr Jeannette Chin

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This report documents the design, implementation, and evaluation of a web-based machine learning recommendation system for Steam, an online distribution platform primarily for video games. The system aims to address the limitations of Steam's existing tag-based recommendation system by using implicit feedback such as playtime, completion rates, and game metadata to generate personalised recommendations for a user. A content-based recommendation system using cosine similarity was developed and integrated with Steam's API, allowing for real-time updates to a user's tastes. This was then deployed through a full-stack architecture consisting of a React.js frontend, Python Flask backend, and PostgreSQL database. The system also supports users without a Steam account by allowing them to select games from the database to build themselves a profile. The system was thoroughly tested, and successfully builds upon Steam's existing recommendation system to provide improved game suggestions to users.

Acknowledgements

I would like to extend a huge thank you to everyone that has supported me over the course of this project, especially my supervisors. Jason, your feedback through the first semester was crucial in shaping the project's direction, whilst Jeannette's amazing support throughout the second semester ensured the project's successful completion. Thank you both so much! I also want to say a huge thank you to my family and friends for their encouragement and support over the course of the year.

Contents

1. Introduction **6**
1.1. Project Description 6
1.2. Motivation 6
1.3. Aims and Objectives 7
1.4. Scope of Project 8

2. Background **9**
2.1. Context 9
2.2. Literature Review 9
2.2.1. Types of Recommendation Systems 9
2.2.2. Machine Learning Approaches 10
2.2.3. Relevance to the Project 11
2.3. Existing Systems Analysis 11

3. Preparation **12**
3.1. Dataset 12
3.1.1. Existing Datasets 12
3.1.2. Dataset Creation 13
3.1.3. Dataset Statistics 15
3.2. Algorithm Experimentation 15
3.2.1. Experiment Introduction 15
3.2.2. Overview of Tested Approaches 16
3.2.3. Experiment Evaluation 17
3.2.4. Experiment Results 17
3.3. Model Selection 19

4. Design **19**
4.1. Technology Stack Selection 19
4.1.1. Frontend Technology 19
4.1.2. Backend Technology 20
4.1.3. Database Technology 20
4.2. System Architecture Design 21
4.3. Database Schema Design 22

4.4. Recommendation System Design 24
4.4.1. Feature Vector Design 24
4.4.2. User Profile Generation 25
4.5. User Interface Design 25
4.5.1. User Flow Design 26
4.5.2. Website Design 26
4.6. API Design 26
4.6.1. API Endpoint Design 27
4.6.2. Steam API Integration Design 27

5. Implementation **27**
5.1. Database Implementation 28
5.2. Backend Implementation 28
5.2.1. Steam API Integration 28
5.2.2. API Endpoints 28
5.3. Recommendation System Implementation 29
5.3.1. User Profile Creation 29
5.3.2. Achievement-Based Weighting 30
5.3.3. Recommendation Generation 30
5.3.4. No-Steam Recommendation Approach 31
5.4. Frontend Implementation 31
5.4.1. Landing Page 31
5.4.2. Loading Page 31
5.4.3. Recommendations Page 32
5.4.4. No-Steam Page 32
5.5. Implementation Challenges and Solutions 32
5.5.1. Steam API Rate Limiting 32
5.5.2. Performance Optimisation 32

6. Testing **33**
6.1. Frontend Unit Testing 33
6.2. Backend API Testing 34
6.3. Performance Testing 34
6.3.1. Response Time Analysis 34

6.3.2. Impact of Achievement-Based Weighting 35
6.3.3. Scalability Assessment 35
6.4. Testing Conclusions 36

7. System Evaluation **36**
7.1. Recommendation Quality Evaluation 37
7.2. Evaluation Summary 37

8. Conclusion and Future Work **37**
8.1. Findings 38
8.1.1. Effectiveness of Content-Based Filtering 38
8.1.2. System Performance Insights 38
8.2. Key Achievements 38
8.3. Limitations and Scope for Improvement 39
8.3.1. Dataset Limitations 40
8.3.2. Algorithm Trade-offs 40
8.3.3. Technical Constraints 40
8.4. Future Work 41
8.4.1. Enhanced Dataset and Features 41
8.4.2. Algorithm Improvements 41
8.4.3. Technical Improvements 42
8.4.4. General Improvements 42
8.5. Concluding Statement 42

References **44**

A. Wireframes, Diagrams, Designs **46**

B. Code Snippets **51**

C. Test Results **59**

1. Introduction

This section provides an introduction to my final year university project titled *Implementation of a Machine Learning Recommendation System*. It outlines the project description, motivation, aims and objectives, and scope.

1.1. Project Description

The project involves the design and implementation of a web-based platform that delivers personalised video game recommendations to users based on the catalogue of games available on Steam. Steam, developed by Valve Corporation, is a leading digital distribution platform and marketplace for video games, offering a vast library of titles alongside social networking features such as user reviews, forums, and community groups (Valve Corporation (2025a)). The proposed recommendation system will utilise machine learning to analyse user preferences, game metadata, and behavioural data to suggest games that align with individual user interests. The platform will feature a user-friendly interface, allowing users to interact seamlessly with the recommendation engine, view game details, and use the platform even if they do not own a Steam account.

1.2. Motivation

The motivation for this project stems from the limitations observed in Steam's existing recommendation system, which primarily relies on user-defined tags to categorise and suggest games. Having used Steam for over ten years, the shortcomings of the current system were obvious to me. While tag-based systems are simple and effective to an extent, they often fail to capture user preferences or user engagement patterns. This project aims to address these shortcomings by developing a machine learning-based recommendation system that uses a wider range of contextual data, including user play history, game genres, ratings, and user engagement. The goal is to create a more accurate and personalised recommendation engine, providing higher quality recommendations for Steam users.

1.3. Aims and Objectives

The primary aim of this project is to design, implement, and evaluate a machine learning-based recommendation system that provides high-quality, personalised video game recommendations to Steam users, compared to the platform's existing tag-based system. The aims of this project are as follows:

- Develop a web-based recommendation platform that integrates with Steam's game catalogue.
- Utilise machine learning algorithms to deliver personalised game recommendations based on user preferences and contextual game data.
- Evaluate the effectiveness of the recommendation system in terms of accuracy, user satisfaction, and performance compared to Steam's existing system.

To achieve these aims, the project will adhere to the following objectives, prioritised using the MoSCoW framework (Must Have, Should Have, Could Have, Won't Have):

Must Have:

- Implement a functional web-based platform with a user interface for browsing and receiving game recommendations.
- Develop a machine learning model that processes user data and game metadata to generate recommendations.
- Integrate the system with Steam's API to access game data, such as titles, genres, and user ratings, as well as real-time user information.
- Conduct testing to ensure the system is reliable, secure, and scalable for a moderate user base.

Should Have:

- Incorporate additional contextual data for improved recommendations.
- Allow a way for users without a Steam account to use the platform.

- Support multi-platform accessibility, ensuring the web platform is responsive on mobile and desktop devices.

Could Have:

- Implement a feedback mechanism allowing users to rate recommendations, enabling continuous improvement of the machine learning model.
- Provide users with options to customise their recommendation preferences (e.g., prioritising specific genres or gameplay styles).
- Explore hybrid recommendation approaches combining collaborative and content based filtering for enhanced accuracy.

Won't Have:

- Real-time multiplayer features, as these are beyond the scope of the recommendation system.
- Support for non-Steam game platforms, as the project focuses exclusively on Steam's ecosystem.
- Advanced user profile management features, such as social integration, due to time constraints.
- Security features, as only publicly available data is being processed and displayed.

1.4. Scope of Project

The scope of the project focuses on the development of a web-based recommendation system tailored to the Steam platform, using machine learning to provide personalised game suggestions. The system will focus on processing structured data (e.g., game genres, user playtime, and ratings) to generate recommendations. The project will include the design of a front-end interface for user interaction, a back-end infrastructure for data processing and model deployment, and integration with Steam's API for data retrieval. The scope is limited to a single-user recommendation system, meaning it will not include features for group recommendations or social networking capabilities. Exclusions include support for non-Steam platforms, or advanced features requiring significant computational resources beyond the project's constraints.

2. Background

This section provides a background for the project by outlining the context of Recommendation Systems (RS) in the gaming industry, providing a brief literature review, and analysing existing recommendation systems on major gaming platforms.

2.1. Context

RS have become integral to digital platforms, especially across industries like e-commerce, content streaming services, and gaming, where they enhance the user experience by delivering personalised content. In the gaming industry, platforms such as Steam, Xbox Store, PlayStation Store, and Nintendo eShop use RS to suggest games that align with users' preferences in hopes of increasing engagement and sales. Steam, the primary focus of this project, is the biggest digital distribution platform for PCs, hosting over 100,000 games (Valve Corporation (2025a)). The gaming industry presents unique challenges for RS, including subjective user tastes, minimal user interaction data, and the need for scalability to handle large game catalogues.

2.2. Literature Review

The literature review conducted for this project provides a critical analysis of recommendation systems and their application in the gaming industry.

2.2.1. Types of Recommendation Systems

According to Gatzioura (2015), RS aim to deliver meaningful content suggestions to active users. The three primary types of RS are Collaborative Filtering (CF), Content Based Filtering (CBF), and Hybrid Models, each with distinct strengths and weaknesses relevant to this project.

- **Collaborative Filtering:** Assumes that users with similar past preferences will share future interests (Koren (2008a)). It is divided into user-based CF, which predicts preferences based on similar users, and item-based CF, which compares items based on user ratings. Latent factor models, such as matrix factorisation (e.g. Singular Value Decomposition (SVD) or Alternating Least Squares (ALS)), enhance CF by reducing the user-item interaction matrix into lower-dimensional

factors (Koren (2009)). CF is effective for capturing hidden relationships without requiring item metadata, making it suitable for gaming where user preferences can be diverse. However, CF struggles with subjective tastes in gaming, as overlapping interests (for example someone liking both action and strategy games) may not guarantee preference for hybrid genres (Koren (2009)).

- **Content-Based Filtering:** Recommends items based on features of previously liked items, using techniques like TF-IDF or cosine similarity to compute item similarity (Lops et al. (2011)). In gaming, features might include genres, mechanics, or tags. CBF is better for new users with sparse data, as it relies on item metadata rather than user interactions (Pazzani and Billsus (2007)). However, it risks limiting recommendations to familiar items, reducing diversity, and depends heavily on accurate metadata, which can be problematic, as seen with Steam's user-generated tags. (Lops et al. (2011)).

- **Hybrid Models:** Combine CF and CBF to mitigate their individual limitations, improving accuracy and diversity (Gomez-Uribe and Hunt (2015)). Platforms like Netflix use hybrid approaches to integrate user interactions and item metadata (Gomez-Uribe and Hunt (2015)). In gaming, hybrid models can address CF's subjectivity issues and CBF's lack of diversity. However, their complexity and computational demands may pose challenges within the project's scope (Schafer et al. (2007)).

2.2.2. Machine Learning Approaches

Machine learning can enhance RS by modelling complex user behaviour. Deep learning models, such as Neural Collaborative Filtering (NCF), excel at capturing non-linear relationships but are computationally expensive, which limits their feasibility for this project (He et al. (2017)). Reinforcement Learning (RL), focusing on long-term user engagement, adapts recommendations based on feedback but requires substantial data, potentially challenging given the project's scale (Afsar et al. (2022)). For this project, simpler models like ALS, which perform well with implicit feedback like playtime, would be more practical, with potential exploration of hybrid approaches if resources allow (Koren (2008b)).

2.2.3. Relevance to the Project

The literature highlights the suitability of CBF and Hybrid Models for gaming RS, particularly when leveraging implicit feedback like playtime, which aligns with Steam's API capabilities (Cheuque et al. (2019)). The project will prioritise CBF for its ability to utilise game metadata. Challenges like data sparsity and subjective tastes, highlighted in the literature, will be addressed through strategies like user taste profiling for non-Steam users and incorporation of implicit feedback (Lops et al. (2011); Koren (2009)).

2.3. Existing Systems Analysis

Several gaming platforms already employ RS to recommend games, each with unique approaches and limitations. Table 1 analyses the recommendation systems of Steam, Xbox Store, PlayStation Store, and Nintendo eShop to identify best practices and areas for improvement, informing the design of this project's RS.

Platform	**Approach**	**Strengths**	**Weaknesses**
Steam	CBF	Simple, effective for broad categorisation; promotes trending games	Inconsistent or sparse tags; lacks depth in capturing complex preferences; prioritises popular games
Xbox Store	Hybrid	Leverages cross-platform user behaviour; uses explicit feedback	Biased toward Game Pass titles; limited diversity for niche genres
PlayStation Store	Hybrid	Tailors recommendations with contextual data; focuses on high-quality games	Overemphasis on first-party titles; recommendations may feel repetitive
Nintendo eShop	CBF	Intuitive for casual users; promotes indie titles	Limited personalisation; recommendations may not evolve with user preferences

Table 1: Comparison of Game Platform Recommendation Systems

By analysing these systems, the project identifies the need for a RS that combines the simplicity of CBF with the personalisation of adding extra implicit data to strengthen recommendations, tailored to Steam's diverse catalogue. The proposed system will address common weaknesses, such as tag inconsistency and lack of diversity, by leveraging machine learning to process implicit feedback and metadata, delivering a more robust and user-centric recommendation experience.

3. Preparation

This section aims to detail the steps taken to prepare for this project, primarily exploring the creation of the dataset used for this project, and carrying out an experiment to determine the RS approach for this project.

3.1. Dataset

This section details the creation and design of the dataset used in the experiment, and the overall project.

3.1.1. Existing Datasets

Several existing datasets containing Steam games were available on platforms such as Kaggle, including 'Steam Games Dataset' (Tomashvili (2024)). Whilst datasets such as these did provide valuable data, they often lacked specific features needed for this project, such as up-to-date player activity, review information and detailed game metadata, as well as not being up to date with the Steam store's offerings at the time of this project.

I chose to create my own dataset to tailor it specifically to the needs of my project. Existing datasets had incomplete, outdated, or insufficient features, making them unsuitable for an accurate and up-to-date recommendation system. By building my own dataset, I ensured data integrity, relevance, and quality, which were crucial for the accuracy of my experiments. Additionally, having full control over the dataset allowed me to better align it with the project's goals.

3.1.2. Dataset Creation

To design the dataset, I focused on collecting features that could best identify similar games based on their metadata. The key attributes I wanted to gather included:

- Game title
- Genre(s)
- Categories
- Number of active players
- Positive review ratio

These features were chosen to ensure the dataset could support robust similarity measurements, which are vital for providing accurate game recommendations.

To create the dataset, I used the Steam API, which is publicly available following a simple registration process. The API provides an endpoint that lists every app available for purchase on Steam, which served as the foundation for the dataset. At the time of creation, this list contained over 130,000 elements, an example element from the list can be seen below:


```
{"appid":10,
"name":"Counter-Strike",
"last_modified":1729702322,
"price_change_number":26391044}
```


The important data in each element of the foundational dataset is the AppID, which is an identifier assigned to each application hosted on steam.

Due to the Steam API's limit of 100,000 API calls per day, it was not feasible to retrieve metadata for every game in a single batch. To resolve this, I decided to create a dataset of reduced size, which also helped solve another challenge. This being that Steam lacks strong moderation for games being listed for sale on the store-front, which has led to many low quality games, alongside games with inappropriate content, being available on the Steam store. The vast majority of these games have few players or poor reviews. It can be assumed that few users would want to be recommended these games, so to address this I introduced filtering prior to collecting game metadata and adding an

app to the final dataset. This reduced the number of API calls made, sped up the dataset creation process, and also ensured a high quality dataset.

The initial filtering process involved two main criteria:

1. **Review Score:** A game must have at least 70% positive reviews. This was determined using a store page API endpoint, which provides counts of total, positive, and negative reviews. The popularity score was calculated as:

   ```
   pop_score = num_positive_reviews / num_total_reviews
   ```

2. **Active Players:** A game must have at least 25 players online during peak hours. This data was retrieved from the same API endpoint, which reports the number of active players at the time of the API call. Peak activity times were derived from data obtained through SteamCharts (SteamCharts (2024)). This number of active players was decided after consulting SteamCharts, and realising that if the active players value was set too high, the dataset would be too small. At peak times, less than 1000 games had more than 100 active players. Having 25 active players implies that there is still a community who values the experience the game provides, and as such would make it an appropriate recommendation for a user.

A script was written to iterate through each AppID in the list of all apps on Steam. Games that did not meet the popularity or activity criteria were skipped. If both criteria were met, the script made an API call to retrieve the game's metadata, including its genres and categories. The final dataset was structured as follows:

```
|AppID|Title|Genre(s)|Categories|Active Players|Review Score|
```

Using this filtering approach ensured that only high-quality, relevant games were included in the dataset, improving the accuracy of subsequent recommendations. Once the dataset creation script was run, it reduced the number of games in the dataset from over 130,000 to just 3,735, which speaks to the amount of poor quality applications currently being hosted on the Steam Store.

3.1.3. Dataset Statistics

Information about the final dataset's contents can be seen in Table 2 below.

Metric	Value
Total Games	**3,735**
Unique Genres	**28**
Unique Categories	**102**
Average Positive Review Ratio	**86.5%**
Average Active Players	**1,512**

Table 2: Dataset Contents

3.2. Algorithm Experimentation

This section details the steps taken to decide which Recommendation System approach to use for the final system.

3.2.1. Experiment Introduction

The purpose of this experiment is to evaluate various recommendation system approaches and determine the most effective method for suggesting similar games based on genres, categories, review scores, and player activity. The key objectives are:

- To compare the performance of several recommendation techniques.
- To measure and analyse the accuracy (percentage of relevant recommendations) and execution speed of each approach.
- To identify the model that performs the best on the given dataset.

The experiment will use three test games to return recommendations for. The three test games used were:

- **Counter-Strike:** A multiplayer FPS (First-Person Shooter).
- **Half-Life:** A single-player FPS (First-Person Shooter).

- **Dota 2:** A MOBA (Multiplayer Online Battle Arena).

These games were selected for their varied genres, categories, and their relevance to one another (e.g., Counter-Strike and Half-Life share similar mechanics and the same development engine, while Dota 2 is more niche). This allowed for a comprehensive evaluation of the different approaches' ability to provide relevant recommendations.

3.2.2. Overview of Tested Approaches

Several approaches were considered for the RS, including pre-built Python libraries such as Surprise (Surprise (2024)). However, due to the dataset's structure, Collaborative Filtering was not feasible for this experiment. Therefore, the following recommendation techniques were explored:

Approach	**Type**	**Description**
Cosine Similarity	Content-Based	Measures similarity between games using vectorisation of genres and categories.
kNN	Content-Based	Finds nearest games in high-dimensional space based on feature similarity.
Random Forest	ML Model	An ensemble learning method that builds multiple decision trees for classification and recommendation.
TensorFlow	Deep Learning	Utilises deep learning models to learn game similarities and provide recommendations.
XGBoost	ML Model	A gradient boosting technique that leverages game metadata for high-accuracy recommendations.
Transformer Embeddings	Deep Learning	Uses Sentence Transformers to convert game metadata into dense vectors for similarity detection.

Table 3: Recommendation Techniques and their Descriptions.

3.2.3. Experiment Evaluation

Each approach was evaluated based on two primary metrics:

- **Speed:** The time required to generate recommendations.
- **Relevance:** The percentage of recommendations deemed relevant based on game genre similarity and user feedback.

3.2.4. Experiment Results

The results for the time taken to generate recommendations were as follows:

Approach	Counter-Strike	Half-Life	Dota 2	Average
Cosine Similarity	0.3633s	0.3857s	0.3705s	**0.3731s**
kNN	0.0010s	0.0010s	0.0010s	**0.0010s**
Random Forest	0.0140s	0.0120s	0.0140s	**0.0133s**
TensorFlow	0.0945s	0.0505s	0.0500s	**0.065s**
XGBoost	0.0020s	0.0010s	0.0010s	**0.0013s**
Transformer Embeddings	10.84s	11.32s	10.56s	**10.90s**

Table 4: Query Times for Three Games Using Each Approach

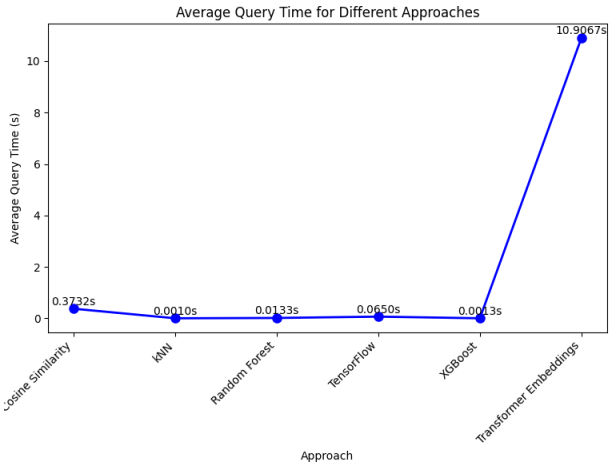

Figure 1: Average Query Time for Each Approach Across Three Games.

The results indicate that the **kNN** and **XGBoost** methods were the fastest, with **Transformer Embeddings** being the slowest due to the complexity of deep learning models.

The relevancy scores for each approach and game were determined manually through a basic sanity check. Five recommended games were returned for each test game and approach, and then a manual check was done to see how relevant each returned game was to the original game. The relevancy percentage of recommendations for each approach is summarised below:

Approach	Counter-Strike	Half-Life	Dota 2	Average
Cosine Similarity	80%	60%	60%	**66.66%**
kNN	60%	40%	40%	**46.6%**
Random Forest	60%	60%	40	**53.3%**
TensorFlow	80%	40%	20%	**46.6%**
XGBoost	80%	60%	20%	**53.3%**
Transformer Embeddings	60%	60%	20%	**46.6%**

Table 5: Relevancy percentage for three games for each approach.

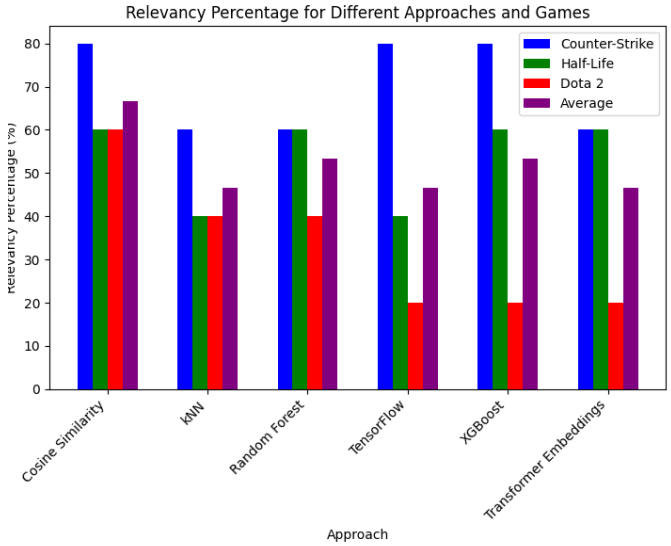

Figure 2: Relevancy percentage for each approach across three games.

Upon reviewing the results, they align with expectations. Counter-Strike had the largest number of similar games, reflecting its popularity and influence within its genre. In contrast, Half-Life, while still a popular title, has fewer similar games on Steam compared to Counter-Strike, which is consistent with its single-player story-driven gameplay. Finally, Dota 2, being a niche Multiplayer Online Battle Arena (MOBA) with a distinct gameplay style, has fewer relevant recommendations, which accounts for the lower number of similar games returned for that query.

3.3. Model Selection

Based on the results of the above experiments, the final approach chosen was a cosine similarity-based recommendation model. On average, it provided the most relevant recommendations, and although it is not the fastest, the slightly slower processing speed is a worthwhile trade-off for the higher quality recommendations that the approach provides.

4. Design

This section outlines the design decisions and architectural considerations decided for the project prior to implementation.

4.1. Technology Stack Selection

Choosing an appropriate technology stack was an important early design decision that would influence the entire development process. To choose the technology stack, I compared a few familiar technologies to decide which would be most suitable to meet the project's requirements.

4.1.1. Frontend Technology

For the frontend, I considered a couple of different frameworks. The first was React.js (Meta Inc. (2024)), which I had a small amount of previous experience with. React uses a component-based architecture, and has a substantial amount of documentation available to help with development. I also considered using Vue.js (Vue.js (2024)), which also has lots of documentation, and has built in state-management. After evaluating the

pros and cons of both, I decided to go with React.js for the frontend for the following reasons:

- **Performance:** The virtual Document Object Model (DOM) provides efficient rendering, which would be important for displaying the dynamic recommendation results. Using a DOM connects web pages to scripts or programming languages by representing the structure of a document
- **Ecosystem:** The ecosystem has native libraries like React Router for navigation and Axios for API requests.
- **Prior Experience:** Existing familiarity with React reduced the learning curve and accelerated development.

4.1.2. Backend Technology

For the backend, the first technology considered was Express.js (Express.js (2025)), which meant that JavaScript would be used throughout the stack. I also evaluated Flask (Pallets Projects (2024)), a python library. I chose Flask as the backend framework for the following reasons:

- **Python Ecosystem:** Python has a wide range of libraries like NumPy, SciPy, and scikit-learn, which will be crucial for the development of the recommendation algorithm.
- **API-First:** Easy to implement APIs with a React frontend, which is the primary backend requirement.
- **Prior Experience:** I had previous experience using Python for machine learning from my first semester Machine Learning module.

4.1.3. Database Technology

For the Database, I considered both SQLite (SQLite (2024)) and PostgreSQL (PostgreSQL Global Development Group (2024)), but decided on PostgreSQL for the following reasons:

- **Array Support:** Native support for array data types, which is ideal for storing game genres, categories, and feature vectors.

- **Scalability:** Ability to handle the expected dataset size easily, and scale in the future if required.
- **Integration:** Strong integration with Python and Flask via the psycopg2 library (Di Gregorio (2023)).
- **Familiarity:** I had significant previous experience with PostgreSQL from previous course modules.

4.2. System Architecture Design

The system was designed following a three-tier architecture pattern, to separate the presentation layer, application layer, and data layer. This approach will help with modularity, maintainability, and scalability.

The architecture will consist of the following components:

- **Frontend (Presentation Layer):** React.js application that provides the user interface for interacting with the recommendation system. Communicates with the backend via HTTP/JSON.
- **Backend (Application Layer):** Flask application that processes requests from the frontend, interacts with the database, and integrates with the Steam API. Contains three main components:
  - **API Endpoints:** Handles HTTP requests and responses.
  - **Recommendation Engine:** Implements the content-based filtering algorithm.
  - **Steam API Integration:** Manages communication with the Steam API.
- **Database (Data Layer):** PostgreSQL database that stores game metadata and precomputed feature vectors.
- **External Services:** Steam API for retrieving user data and game information.

A visual representation of the system architecture design can be seen in Figure 3.

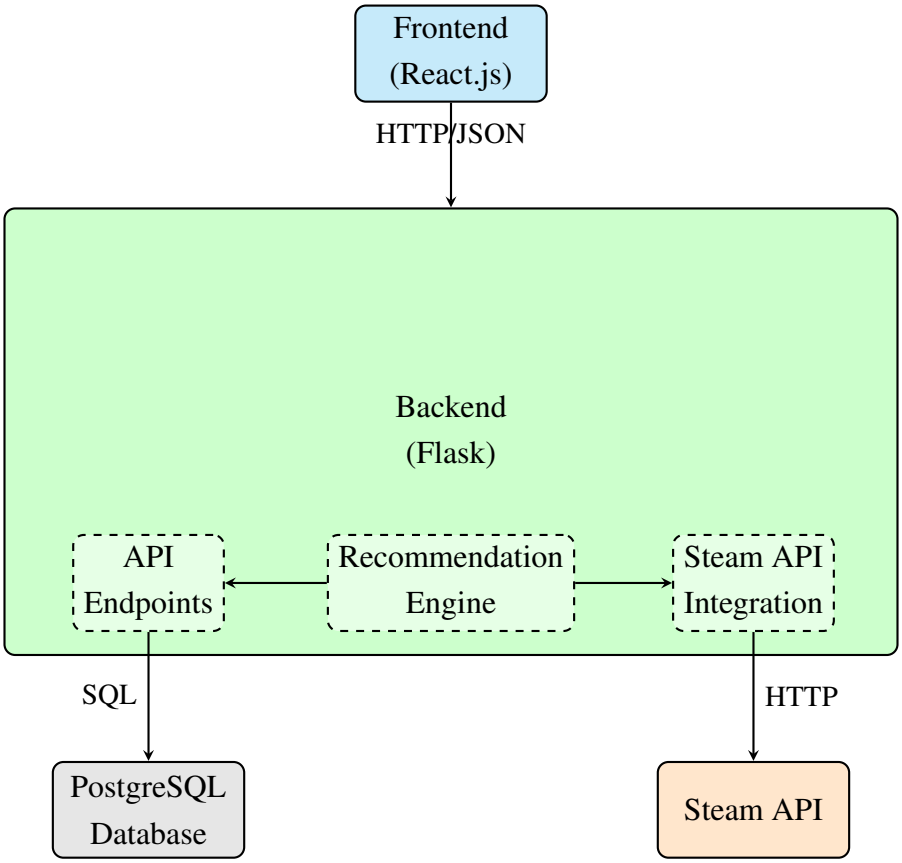

Figure 3: System Architecture Diagram

4.3. Database Schema Design

The database schema was designed to efficiently store game metadata and support the recommendation algorithm's requirements, and is a single table database. A critical design decision was to pre-calculate and store feature vectors for each game, rather than generating them during recommendation requests. The decision to pre-calculate vectors will improve response times by shifting computational work from request-time to database setup. This trade-off prioritises user experience at the expense of the database taking marginally longer to set up, with database creation times seen in Table 6. However, the additional time taken is insignificant, so it is worth doing. Although the database does not refresh automatically, new games are added to Steam every day so an automatic database update script would be a useful future feature.

Approach	**Time Taken (s)**
Database setup (No precomputed game vectors)	**1.2**
Database setup (Precomputed game vectors)	**6.7**

Table 6: Database Creation timings.

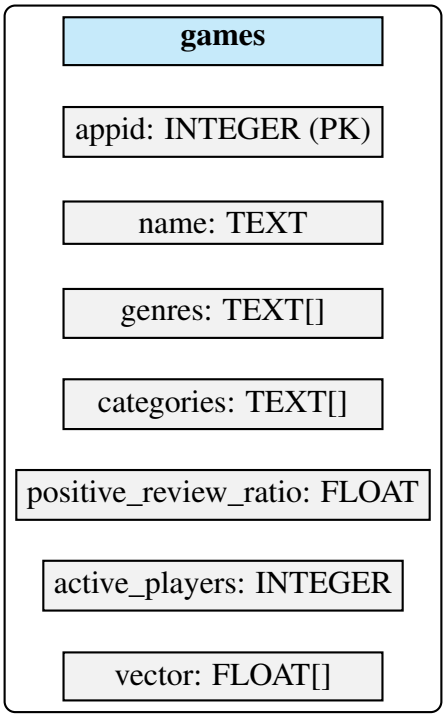

Figure 4: Database Schema for the `Games` Table

The schema will use PostgreSQL's array types to store:

- **genres**: An array of genre names associated with the game.
- **categories**: An array of gameplay categories.
- **vector**: A pre-computed feature vector used for similarity calculations.

This design offers several advantages:

- **Query Efficiency:** Retrieving game vectors requires only a single query.

- **Storage Efficiency:** Arrays eliminate the need for additional junction tables.
- **Performance:** Pre-calculated vectors significantly reduce processing time during recommendation generation.
- **Simplified Logic:** The backend can focus on vector operations rather than feature extraction.

The schema design can be seen in Figure 4 above.

4.4. Recommendation System Design

The recommendation system was designed after careful consideration of various approaches. While Collaborative Filtering is often used in recommendation systems, the Content-Based approach was selected due to its alignment with the project requirements and available data. Additionally, as highlighted in Section 2.3, this approach is already widely used within the industry, so it made sense to use it and try to expand on it.

4.4.1. Feature Vector Design

The design of feature vectors is crucial for Content-Based recommendation systems (Pazzani and Billsus (2007)). The vector representation for each game was carefully crafted to capture relevant characteristics, with One-hot encoding used to represent both the genre and category features. One-hot encoding is a technique that converts categorical data into a binary format, where each category is represented by a unique binary vector with a single '1' and all other values as '0'. When using One-hot encoding to represent a game by its genres/categories, a binary vector is created for each game, with a '1' for each genre the game belongs to (e.g., Action, Adventure) and '0' for genres it does not.

1. **Genre Features:** One-hot encoded representation of game genres (e.g., Action, RPG, Strategy).
2. **Category Features:** One-hot encoded representation of game categories (e.g., Multiplayer, Single-player, VR Support).
3. **Review Score:** Percentage of positive reviews.

4. **Player Count:** Log-transformed active player count to reduce the influence of extremely popular games.

The feature vector design ensures that similarity calculations consider both categorical features (genres, categories) and continuous metrics (review score, player count). Using log transformation for the player count value was implemented to prevent more popular games from having a large weight difference to smaller titles.

4.4.2. User Profile Generation

To generate personalised recommendations, the system will create a user profile vector based on their gaming history. Two approaches were designed:

1. **Playtime-Weighted Profile:** A weighted average of game vectors, with more weight given to games with higher playtime. The weight for a certain game can be calculated as:

$$\text{weighted\_game\_vector}_i = \text{game\_vector}_i \times \left(\frac{\text{playtime}_i}{\text{total\_playtime}} \right)$$

2. **Achievement-Enhanced Profile:** A weighted average incorporating both playtime and achievement completion rates. The weight for a game Calculated as:

$$\text{weight}_i = \frac{\text{playtime\_weight}_i + \text{achievement\_completion\_rate}_i}{2}$$

$$\text{weighted\_game\_vector}_i = \text{game\_vector}_i \times \text{weight}_i$$

These equations are performed for each game the user owns, and are summed to provide a representational vector of the user's taste in games. The dual-weighting approach was designed to provide more accurate representations of user preferences, based on the idea that high achievement completion indicates stronger engagement with a game, regardless of playtime.

4.5. User Interface Design

The user interface was designed with a focus on simplicity and usability. The design process included creating wireframes and mock-ups before implementation.

4.5.1. User Flow Design

A user flow was designed to understand how users might navigate through the recommendation process, as seen in Figure 5

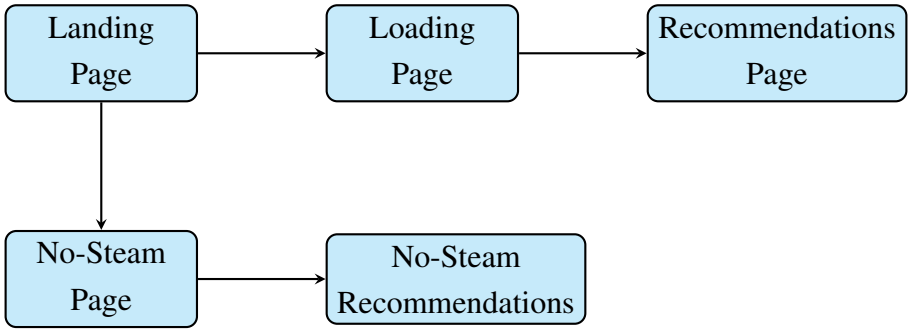

Figure 5: User Experience Flow Diagram

The user flow was designed to accommodate two primary user scenarios:

- **Steam Users:** Enter their username, wait for recommendations to be generated, and view personalised suggestions.
- **Non-Steam Users:** Navigate to the No-Steam page, select games they enjoy, and receive recommendations based on these selections.

This design ensures the system is accessible to both Steam users and non-Steam users to meet one of the key project objectives.

4.5.2. Website Design

The website was designed by creating wireframes of each page. The design for the landing page can be seen in Figure 7, the design for the loading page can be seen in Figure 8, the design for the Recommendations Page can be seen in Figure 10, and the designs for the No-Steam Page can be seen in Figure 9.

4.6. API Design

The backend API was designed following RESTful principles (RESTful API (2024)), with clearly defined endpoints for different functionalities.

4.6.1. API Endpoint Design

The API endpoints designed to support this application can be seen in Table 12.

The API was designed with the following principles:

- **RESTful:** Resources are identified by URLs and accessed using standard HTTP methods.
- **JSON-Based:** All responses are in JSON format for easy consumption by the frontend.
- **Stateless:** Each request contains all the information needed to process it.
- **Error Handling:** Consistent error responses with appropriate HTTP status codes.

4.6.2. Steam API Integration Design

Integration with the Steam API was designed to retrieve essential data while respecting rate limits and handling potential errors. The use of the following API endpoints was decided through consulting the Steam API Documentation (Valve Corporation (2025b)).

Steam API Endpoint	Purpose
ISteamUser/ResolveVanityURL	Convert username to Steam ID
IPlayerService/GetOwnedGames	Retrieve user's game library
ISteamUserStats/GetPlayerAchievements	Get achievement data for a user
ISteamUserStats/GetGlobalAchievementPercentagesForApp	Get achievement data for a specific game

Table 7: Required Steam API Endpoints.

5. Implementation

This section details the implementation of the system, covering the database setup, backend development, recommendation algorithm implementation, and frontend user interface implementation.

5.1. Database Implementation

The database uses the single table schema design in Figure 4, and stores all necessary game metadata for the recommendation system.

A Python script (setup_db.py) was developed to automate the creation and population of the database. This script creates the database if it does not exist, then it creates the **Games** table. After this, it loads the games from the .json dataset, pre-computes the vector representation for each game, and then stores all the necessary information in the database. The code snippet in Figure 15 shows the process of computing the vector for each game. By pre-computing and storing vectors during database setup, the system significantly reduces the computational load during real-time recommendation generation, enabling faster response times for users.

5.2. Backend Implementation

The backend of the system was implemented using Flask. This component sits in between the front-end user interface and the database, while also interfacing with the Steam API.

5.2.1. Steam API Integration

Integration with the Steam API was an important part of the backend implementation. The system interacts with several Steam API endpoints to retrieve user data and game information, outlined in Table 7.

The integration was implemented using Python's **requests** library, which allowed for requests to be made to the API, ensuring the rate limits were respected.

5.2.2. API Endpoints

The backend has several API endpoints to serve the frontend application:

API Endpoint	Purpose
GET /player-data	Retrieves a user's Steam profile and game library
GET /recommendations	Generates personalised game recommendations by using cosine similarity to find the nearest matches to the user vector
GET /games	Retrieves all games from the database
POST /no-steam-recommendations	Generates recommendations based on selected games

Table 8: Backend API Endpoints.

The **/player-data** endpoint, a snippet of which can be seen in Figure 16, captures the user's username, and if needed, resolves it. A call is then made to the Steam API, and a list of the user's owned games and playtime for each is returned.

The **/recommendations** endpoint is the primary endpoint for generating recommendations for users, and is further explained in Section 5.3.

The **/games** endpoint returns all games in the database, and is used for the No-Steam page. A code snippet of this endpoint can be seen in Figure 17.

Finally, the **/no-steam-recommendations** is used to build a user vector for users without a Steam account, and is further explained in Section 5.3.4.

5.3. Recommendation System Implementation

The core Recommendation System uses a Content-Based Filtering approach with cosine similarity to identify games similar to those a user has previously played and enjoyed.

5.3.1. User Profile Creation

For users with Steam accounts, the system builds a profile vector based on their owned games and playtime. This is implemented in the build_user_profile function, which works by first gathering identifying games that are owned by the user and are in the database, then retrieves the game vectors for each game from the database. Weights are

then calculated based on playtime (and achievements if enabled), and a weighted sum of game vectors is then created to represent the user's video game preferences. The final vector is then normalised to ensure consistent similarity comparisons. A snippet showing the build_user_profile function can be seen in Figure 18.

5.3.2. Achievement-Based Weighting

An enhancement to the recommendation system was the implementation of achievement based weighting. This optional feature is controlled via a toggle button on the Landing Page, and modifies the influence of each game in the user's profile based on their achievement completion rate:

1. Retrieve the total number of achievements for a game
2. Determine how many achievements the user has completed
3. Calculate a completion rate (completed/total)
4. Combine this with playtime-based weighting to create a more nuanced profile

This approach offers a more accurate representation of user preferences, as games with high completion rates likely indicate stronger user interest regardless of playtime. The process to generate a user vector weighted by achievements and playtime can be seen in Figure 19.

5.3.3. Recommendation Generation

Once a user profile is created, the system generates recommendations by calculating similarity scores between the user vector and all game vectors in the database:

The implementation leverages scikit-learn's cosine similarity function to efficiently calculate similarity scores (scikit-learn Developers (2025)). The formula for cosine similarity is:

$$\text{cosine\_similarity}(\mathbf{A},\mathbf{B}) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Games are then ranked by similarity, with the top 10 returned as recommendations. The system excludes games the user already owns to focus on discovering new content.

5.3.4. No-Steam Recommendation Approach

For users without Steam accounts, an alternative recommendation method was implemented. This approach allows users to select games they enjoy from a searchable list, rank them, and then receive recommendations based on these selections:

This implementation weights selected games based on their ranking, with higher ranked games having more influence on the resulting recommendations. This approach provides a pathway for new users to discover games similar to ones they already enjoy, even without a Steam account. A snippet of the code to create a user vector for a user without a Steam account can be seen in Figure 20.

5.4. Frontend Implementation

The front-end was implemented using React.js, to provide a responsive and interactive user interface. The application follows a multi-page architecture with distinct views for different user journeys. The frontend consists of four main components, each corresponding to a different page in the user journey. Navigation between components is handled using React Router.

5.4.1. Landing Page

The Landing Page component serves as the main entry point to the website, offering users an interface to enter their Steam username and toggle achievement-based recommendations, or go to the 'No Steam' page if they do not have a Steam account. A screenshot of the landing page can be seen in Figure 11.

5.4.2. Loading Page

The Loading Page component serves as a transitional screen while the system fetches user data and generates recommendations. It is a very simple page, with a loading spinner to demonstrate to the user that the system is working on returning recommendations. Once the data has been loaded, this page automatically redirects to the Recommendations Page. A screenshot of the loading page can be seen in Figure 12.

5.4.3. Recommendations Page

The Recommendations Page component displays the game recommendations retrieved from the backend. The component uses Steam's Content Delivery Network (CDN) to display game banners, enhancing the visual appeal of recommendations without needing to store image assets for each game locally. Each game card includes the game banner, title, similarity score, and a direct link to the game's store page on Steam. A screenshot of the Recommendations page can be seen in Figure 14

5.4.4. No-Steam Page

The NoSteamPage component provides a way for users without a Steam account to use the website. The user is able to search the database for five games that they like, rank them, and these games are then used to build a user vector for which similar games can be returned, using cosine similarity similar to the main Recommendations Page. A screenshot of this page can be seen in Figure 13.

5.5. Implementation Challenges and Solutions

During the implementation process, several challenges were encountered and addressed.

5.5.1. Steam API Rate Limiting

The Steam API imposes a limit of 100,000 requests per day, which could be quickly exhausted when fetching achievement data for users with a large amount of games. To resolve this, the Achievement Based recommendations were made optional to reduce the likelihood of request limit being reached, and are turned off by default.

5.5.2. Performance Optimisation

Computing similarity scores between user profiles and all games in the database could be computationally expensive, especially for users who own a large number of games. To optimise this, the project used precomputed game vectors. Additionally, scikit-learn's cosine similarity function was used as it is already optimised (scikit-learn Developers (2025)), and vectors were normalised to allow for faster similarity calculations.

These optimisations reduced the computational time complexity from O(nm) (where n is the number of games and m is the number of features) to O(n).

6. Testing

The following section outlines the testing methodology, libraries, and processes used to test the entire system. The testing strategy targeted all aspects of the system, and included unit testing for the frontend and backend, integration tests, as well as performance tests.

6.1. Frontend Unit Testing

For front-end component testing, Jest and React Testing Library were selected as the primary testing tools. These frameworks were chosen as they offer a solution for testing React components in a way that mimics how users are likely to interact with the application.
Jest (Jest (2024)) served as the test runner and provided a platform for writing unit and integration tests. React Testing Library (Testing Library (2024)) complemented Jest by focusing on testing components from a user's perspective, testing user interactions rather than implementation details. The testing process involved rendering components in a virtual Document Object Model (DOM), simulating user interactions (such as clicks, form submissions, and navigation), and making sure that all components responded correctly to these interactions. This approach ensured that components not only rendered correctly but also maintained expected functionality throughout the development process. A test plan was created, targeting all key interactions a user would have with the website, as well as ensuring that all components were being displayed correctly.

As shown in Table 14, all frontend component tests were successful. The tests covered key user interactions and edge cases, ensuring that the UI behaves predictably under various conditions. Some additional, more detailed test cases were implemented to check further edge cases. A table detailing these additional cases can be seen in Table 13 in the Appendix. Additionally, a snippet of the testing script can be seen in Figure 21.

The final step of testing the frontend was to evaluate the performance of the website

on other browsers/devices. This was done using the Firefox developer tools which allows selection of a variety of devices, as well as testing the system's functionality on different browsers including Chrome, Safari, and Brave.

These test cases demonstrate the approach taken to ensure frontend functionality. Each component's behaviour was tested under expected conditions, as well as edge cases to make sure everything was completely functional.

6.2. Backend API Testing

API testing was carried out for all backend endpoints created for this project. This testing was done using two approaches: POSTman (Postman Inc. (2025)) for manual and automated API testing, and a custom Python testing script for load testing and performance analysis.

POSTman provided an environment for creating and organising API test cases. Tests were organised into collections that verified both the functional correctness and error handling capabilities of each endpoint. The POSTman tests included validation of response structures, status codes, and data types, as well as tests for edge cases and error conditions. Table 15 summarises the API tests conducted and their results. All endpoints were tested for both successful scenarios, error conditions, and edge cases to ensure robust error handling. The tests verified not only status codes but also response structure and content, ensuring that the API returns data in the expected format. A snippet of the POSTman testing script can be seen in Figure 22, and an example of the test script output can be seen in Figure 23.

6.3. Performance Testing

Performance testing was conducted to measure the system's responsiveness under various load conditions.

6.3.1. Response Time Analysis

Response times were measured for key API endpoints using a custom testing script, written in python and using the 'requests' library. Tests were performed with different user profiles and configurations to understand the system's performance characteristics.

These tests were not performed on the No-Steam endpoint, as it only takes five games as input and does not change based on a provided user account. Instead, the endpoint was called with separate games and performed consistently across all tests, due to operating on a fixed number of five games. The results in Table 9 show that response times scale

Endpoint	Small Library (10–50 games)	Medium Library (51–150 games)	Large Library (151+ games)
/player-data	0.32s	0.59s	0.86s
/recommendations (without achievements)	0.58s	0.91s	1.32s
/recommendations (with achievements)	1.78s	3.25s	5.41s

Table 9: API Endpoint Response Times

linearly with the size of a user's game library, particularly when achievement data is included. The achievement-based recommendation endpoint exhibits significantly longer response times due to the additional API calls required to fetch achievement data for each game.

6.3.2. Impact of Achievement-Based Weighting

Analysis was done to understand the performance impact of enabling achievement based weighting. Figure 6 shows the difference in response times for five separate users, with achievements toggled on and off for each user. The users used for testing had a varying amount of games in their account.

As expected, requests with achievement data took longer to process due to the additional API calls required for each game in the user's library, scaling with the number of owned games.

6.3.3. Scalability Assessment

To test how well the system might be able to scale, a Python script utilising the 'requests' library was developed. This script allowed for load testing, measuring response times and success rates under various levels of concurrent requests. When running this script, it simulated a different number of concurrent users, each submitting a request to the recommendations endpoint without achievements enabled. The stress test results in Table 10 indicates that the system maintains good performance with up to 10 concurrent users, with only minor degradation. Beyond this point, performance decreases significantly,

Concurrent Users	Success Rate	Avg. Response Time	Steam API Errors
1	100%	0.94s	0%
5	100%	1.12s	0%
10	98%	1.87s	2%
20	85%	3.41s	12%
50	64%	6.83s	31%

Table 10: Stress Test Results

likely due to Steam API rate limiting, which begins to cause errors in request processing. In this experiment, a failure was considered when the API timed out following 60 seconds without a response. Because of this, the recommended maximum concurrent users is 10 with the current implementation of the website, as a 2% failure rate is acceptable. This suggests that the current implementation is suitable for personal or small-scale deployment. For larger-scale deployments, additional optimisations would be required, possibly including a queue system during peak times, or gathering more API keys to make more calls.

6.4. Testing Conclusions

The approach implemented for testing the system ensured a high-quality, reliable application. By testing the entire system across all application layers, the testing process identified and addressed potential issues before they affected end users. The performance testing results demonstrate that the application can handle the expected user load while maintaining responsive interaction, and the Steam API integration tests confirm reliable communication with external services.

7. System Evaluation

Following the implementation of the system, an evaluation was conducted to assess the quality of recommendations provided. This section presents the evaluation methodology and results.

7.1. Recommendation Quality Evaluation

The quality of recommendations was evaluated using objective metrics. To do this, a test set of 10 Steam users with different gaming preferences was used. For each user, 20% of their games were randomly held out, and the system generated recommendations based on the remaining 80%. The quality was measured by how many of the held-out games appeared in the top 10, 20, and 50 recommendations.

Recommendation Approach	% of games in Top 10	% of games in Top 20	% of games in Top 50
Playtime-only	18%	22%	31%
Achievement-weighted	24%	29%	37%

Table 11: Recommendation Precision

The results in Table 11 show that the achievement-weighted approach outperforms the playtime-only approach across all precision metrics.

7.2. Evaluation Summary

The evaluation results demonstrate that the system successfully provides high quality recommendations. Objective metrics indicate high-quality, relevant recommendations, even more so when using the achievement-weighted approach. This demonstrates that incorporating implicit feedback beyond playtime significantly improves recommendation quality. While this approach does incur a performance cost, the improved relevance justifies the trade-off for users seeking the most personalised and high quality recommendations.

The No-Steam approach does not provide recommendations that are as personal due to the lack of implicit feedback, but still provides recommendations and effectively serves users without Steam accounts. This meets an important project objective by making the system accessible to a broader audience.

8. Conclusion and Future Work

This section summarises the key findings, accomplishments, and limitations of the project, and outlines potential directions for future development and enhancement.

8.1. Findings

The development of the system highlighted several important findings about game Recommendation Systems and Content-Based Filtering approaches.

8.1.1. Effectiveness of Content-Based Filtering

The implementation demonstrated that content-based filtering using cosine similarity is an effective approach for video game recommendations. The system successfully used game metadata (genres, categories, review scores, and player counts) to generate relevant recommendations that aligned with users' gaming preferences. Game genres and categories proved to be strong predictors of user preferences, confirming that content based approaches can be effective in the gaming domain. Enhancing this approach with implicit feedback such as playtime and completion rate also further improved the recommendations.

8.1.2. System Performance Insights

The testing of the system provided insights about the scalability and efficiency of the implementation. When using achievements as an additional implicit feedback measure, Steam API rate limiting emerged as the primary scalability concern. Additionally, retrieving achievement data incurred a performance cost, with response times increasing in tandem with the size of the user's game library.

The database proved to be a suitable choice, with fast retrieval of game vectors helping to provide quick recommendation generation speeds. Finally the React.js front-end performed well, and consistently maintained the design across different devices and browser environments.

8.2. Key Achievements

The project successfully achieved its primary aims and the highest priority objectives outlined in the MoSCoW prioritisation framework from the introduction were implemented. All "Must Have" objectives were successfully completed:

- **Functional Web Platform:** A complete web-based system was implemented with a user-friendly interface for receiving and exploring game recommendations.

- **Machine Learning Model:** A content-based recommendation model was developed and enhanced with achievement-based weighting to generate personalised recommendations.
- **Steam API Integration:** The system successfully integrates with Steam's API to retrieve game data and user information in real-time.
- **System Reliability:** Comprehensive testing confirmed the system's reliability, with appropriate error handling and recovery mechanisms.

Additionally, all "Should Have" objectives were completed:

- **Additional Contextual Data:** The system incorporates achievement completion rates as additional context to improve recommendation relevance.
- **Non-Steam User Support:** The No-Steam flow enables users without Steam accounts to receive recommendations.
- **Multi-Platform Accessibility:** The design ensures the application works well on both desktop and mobile devices.

However, only one of the three "Could Have" objectives was implemented:

- **Customisable Recommendation Preferences:** Users can enable or disable achievement based weighting to prioritise recommendation quality or speed.
- **Feedback Mechanism:** This was not implemented due to time constraints.
- **Hybrid Recommendation Approaches:** Attempting a hybrid approach combining collaborative and content-based filtering was not possible for this project due to the dataset not containing user data.

8.3. Limitations and Scope for Improvement

Despite the project's successes, several limitations were identified that could be addressed in future iterations.

8.3.1. Dataset Limitations

The dataset used for recommendations had several limitations that affected the system's performance.

- **Dataset Size:** The filtering process reduced the original dataset from over 130,000 games to 3,735 high-quality titles. Although I believe this had a positive effect on recommendation quality, it potentially excluded niche games with smaller player bases.
- **Quality Bias:** The dataset filtering criteria introduced a bias toward more popular and well-received games, potentially limiting the diversity of recommendations.
- **Creation Limitations:** The dataset represents a snapshot in time and does not automatically update as new games are released or existing games gain popularity, potentially causing the system to miss recent releases.
- **Limited Feature Set:** While genres and categories are useful features, additional features like story themes, art styles, or gameplay mechanics could further enhance recommendation relevance.

8.3.2. Algorithm Trade-offs

The Content-Based Filtering approach had some trade-offs and limitations.

- **Feature Representation Limitations:** Using One-hot encoding of genres and categories treats all features equally, without capturing relationships between similar genres.
- **Cold Start for New Users:** Users with few games or minimal playtime receive less personalised and lower quality recommendations.
- **Popularity Bias:** Despite log transformation of player counts, popular games may still be overrepresented in recommendations due to the filtering process.

8.3.3. Technical Constraints

Several technical limitations affect the system's scalability and performance.

- **Steam API Rate Limits:** The 100,000 API calls per day limit constrains the number of users who can receive achievement-weighted recommendations, which has led to the recommended maximum of 10 concurrent users.
- **Response Time with Achievements:** Fetching achievement data significantly increases response times, especially for users with large game libraries.
- **Database Update Mechanism:** The current implementation lacks an automated system for regularly updating game data and recomputing vectors.

8.4. Future Work

Based on the limitations identified and personal reflection, several areas for future work have been identified to potentially enhance the system.

8.4.1. Enhanced Dataset and Features

Future work could focus on expanding and enriching the dataset.

- **Expanded Dataset:** Develop better filtering criteria that balance quality with inclusivity, potentially incorporating niche games with smaller but passionate player bases. This could be done by filtering by number of reviews, rather than active players. This would allow for previously successful games to be recommended as well.
- **Automatic Dataset Updates:** Implement a scheduled process to automatically update the dataset with new games and refresh metadata for existing games.
- **Advanced Feature Extraction:** Extract additional features from game descriptions, reviews, and media using other techniques such as natural language processing.
- **Time-based Features:** Incorporate release date information and gameplay trends to better capture time-based aspects of gaming preferences.

8.4.2. Algorithm Improvements

Several algorithmic enhancements could address current limitations:

- **Hybrid Recommendation Approach:** Research how to implement a hybrid system that combines Content-Based Filtering with collaborative filtering techniques to improve recommendation diversity. This would however require a complete redevelopment of the dataset and system.
- **Word Embeddings for Genres:** Replace One-hot encoding with word embeddings or learned embeddings that can capture relationships between genres and categories.
- **Deep Learning Models:** Explore neural network-based recommendation models like Neural Collaborative Filtering (NCF) (He et al. (2017)) for potentially improved accuracy.

8.4.3. Technical Improvements

Future technical improvements could address scalability, one way for this to be achieved could be through implementing a system to manage multiple Steam API keys for use on the website, which would allow for a greater daily request limit, and in turn allow for more users to use the website and the enhanced achievements feature.

8.4.4. General Improvements

Several enhancements could potentially improve the experience of using the website.

- **Recommendation Explanations:** Provide transparent explanations to the user as to why specific games were recommended.
- **Feedback Mechanism:** Implement a system for users to rate recommendations, using this feedback to refine future suggestions.
- **Customisation Options:** Allow users to adjust recommendation parameters, such as adding more weight to a certain genre or game, or excluding certain categories.

8.5. Concluding Statement

This project successfully implemented a content-based recommendation system that provides personalised game suggestions based on user preferences and implicit feedback signals. The system demonstrated the effectiveness of Content-Based Filtering

for game recommendations, with achievement-based weighting improving recommendation relevance compared to only using playtime as a form of implicit feedback.
The project achieved all "Must Have" and "Should Have" objectives from the initial MoSCoW prioritisation, of which I am very proud. Whilst the implemented system has limitations related to dataset scope, algorithm biases, and some technical constraints, these limitations could be improved upon in the future by expanding the dataset, researching hybrid recommendation approaches, and potentially introducing user feedback mechanisms. The system evolves alongside user tastes, as increased playtime and achievement completion over time will change the user profile, ensuring that recommendations adapt to changing preferences over time. Getting the chance to work on this project has improved my analytical skills, coding skills, writing skills, and has given me the chance to create a project linked to something I have been interested in for over a decade. It has been a great experience and has taught me lots!

References

Afsar, M. M., Crump, T., and Far, B. (2022). Reinforcement learning based recommender systems: A survey.

Cheuque, G., Guzmán, J., and Parra, D. (2019). Recommender systems for online video game platforms: the case of steam. In *Companion Proceedings of The 2019 World Wide Web Conference*.

Di Gregorio, F. (2023). psycopg2: Postgresql database adapter for python. *https://pypi.org/project/psycopg2/*.

Express.js (2025). Express.js: Fast, unopinionated, minimalist web framework for node.js. *https://expressjs.com/*.

Gatzioura, A / Sànchez-Marrè, M. (2015). A case-based recommendation approach for market basket data.

Gomez-Uribe, C. A. and Hunt, N. (2015). The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*.

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T. S. (2017). Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*. ACM.

Jest (2024). Jest: Delightful javascript testing. *https://jestjs.io/*.

Koren, Yahuda / Bell, R. (2009). Matrix factorization techniques for recommender systems. *Computer (Volume: 42, Issue: 8)*.

Koren, Y. (2008a). Factorization meets the neighborhood: a multifaceted collaborative filtering model.

Koren, Yahuda / Hu, Y. . V. C. (2008b). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining*.

Lops, P., De Gemmis, M., and Semeraro, G. (2011). Content-based recommender systems: State of the art and trends. In *Recommender Systems Handbook*. Springer.

Meta Inc. (2024). React: A javascript library for building user interfaces. *https://reactjs.org/*.

Pallets Projects (2024). Flask: A python micro web framework. *https://flask.palletsprojects.com/en/stable/*.

Pazzani, M. J. and Billsus, D. (2007). Content-based recommendation systems. In *The Adaptive Web*. Springer.

PostgreSQL Global Development Group (2024). Postgresql: The world's most advanced open source database. *https://www.postgresql.org/*.

Postman Inc. (2025). Postman api testing tool. *https://www.postman.com/*.

RESTful API (2024). Restful api: Resources and guidelines for rest api design. *https://restfulapi.net/*.

Schafer, J. B., Frankowski, D., Herlocker, J., and Sen, S. (2007). Collaborative filtering recommender systems. In *The Adaptive Web*. Springer.

scikit-learn Developers (2025). scikit-learn: Machine learning in python. *https://scikit-learn.org/stable/index.html*.

SQLite (2024). Sqlite: A serverless sql database engine. *https://www.sqlite.org/*.

SteamCharts (2024). Steam player activity data. *https://steamcharts.com/*.

Surprise (2024). Surprise: A python library for recommender systems. *https://surpriselib.com/*.

Testing Library (2024). React testing library: Introduction. *https://testing-library.com/docs/react-testing-library/intro/*.

Tomashvili, N. (2024). Steam games dataset. *https://www.kaggle.com/datasets/nikatomashvili/steam-games-dataset*.

Valve Corporation (2025a). Steam store. *https://store.steampowered.com/*.

Valve Corporation (2025b). Steamworks api documentation. *https://partner.steamgames.com/doc/api*.

Vue.js (2024). Vue.js: The progressive javascript framework. *https://vuejs.org/*.

A. Wireframes, Diagrams, Designs

Endpoint	Method	Parameters	Response
/player-data	GET	username	User's Steam ID and game library with playtime
/recommendations	GET	username, use_achievements	Personalised game recommendations based on user's Steam profile
/games	GET	None	List of all games in the database
/no-steam-recommendations	POST	selectedGames (array)	Recommendations based on selected games

Table 12: API Endpoint Design

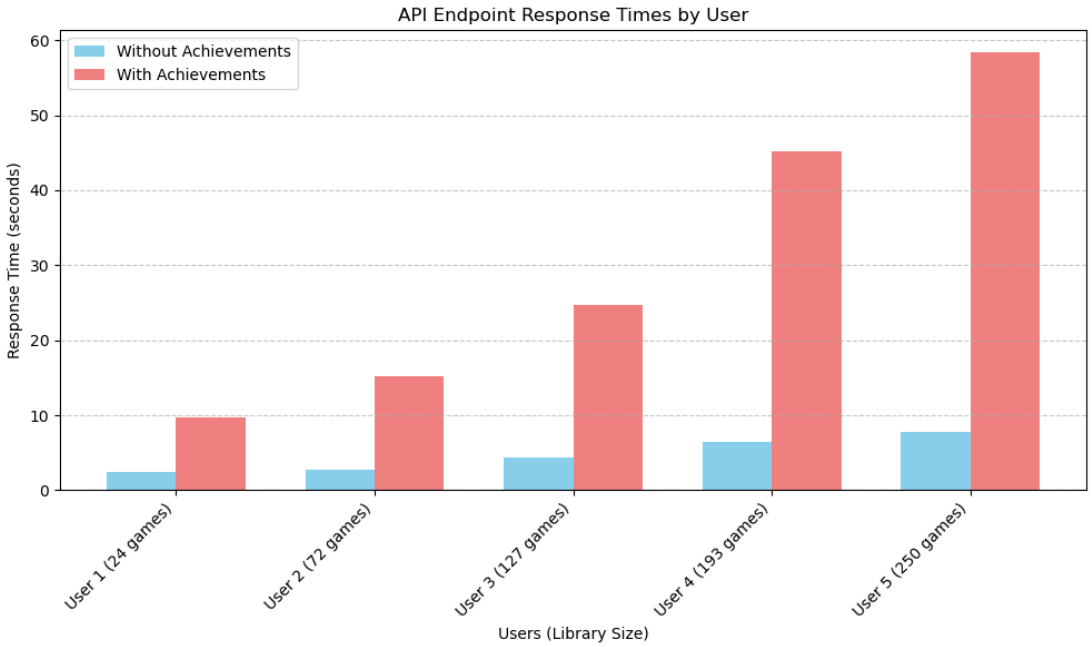

Figure 6: The performance of the Recommendation Endpoint with and without achievements enabled for five users.

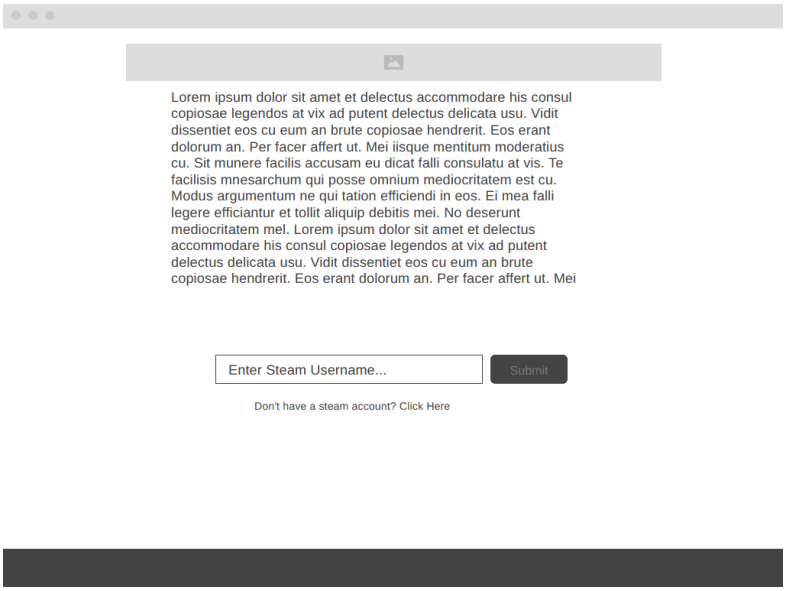

Figure 7: Landing Page Wireframe.

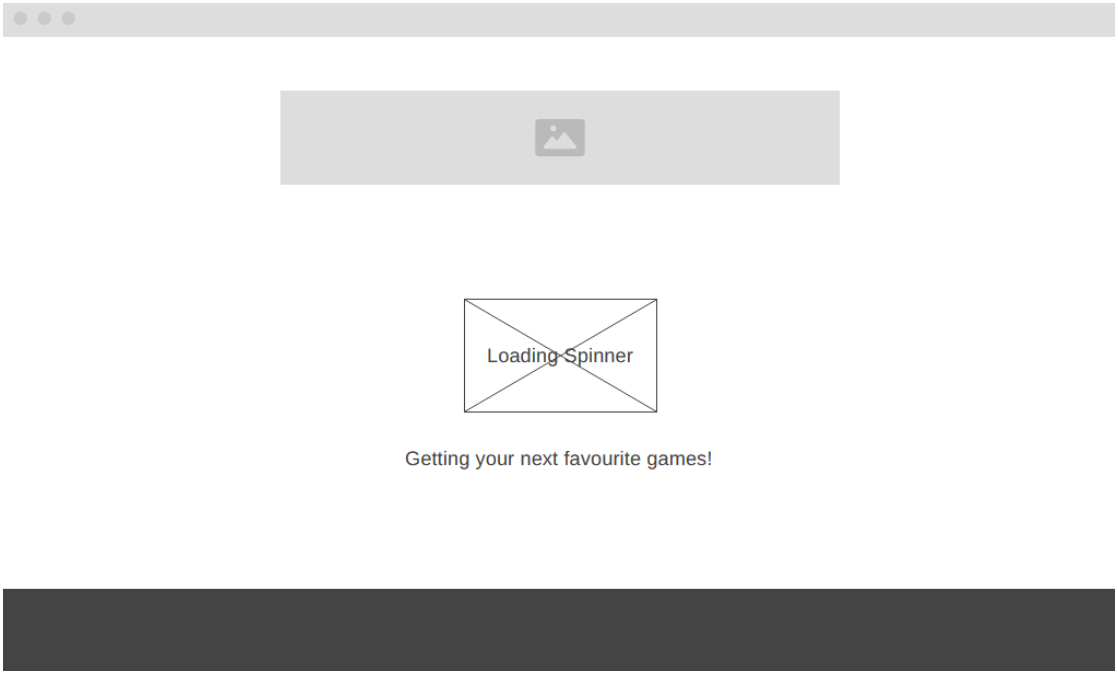

Figure 8: Loading Page Wireframe.

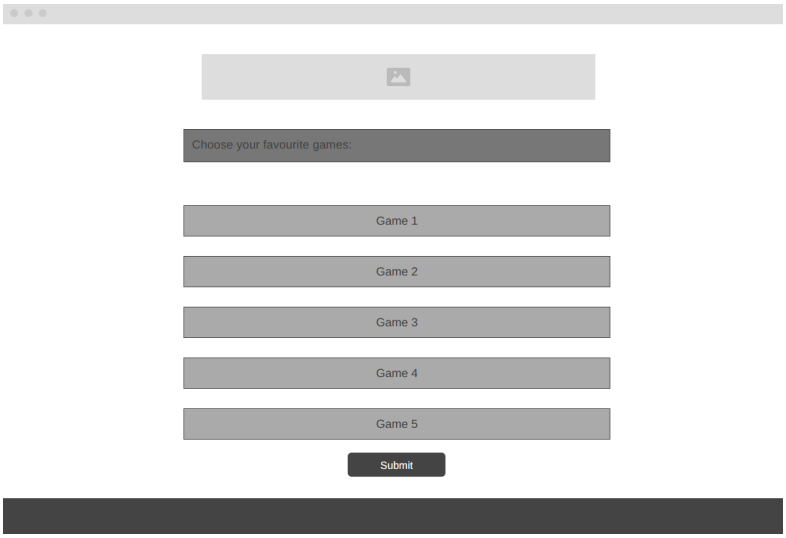

Figure 9: No-Steam Page Wireframe.

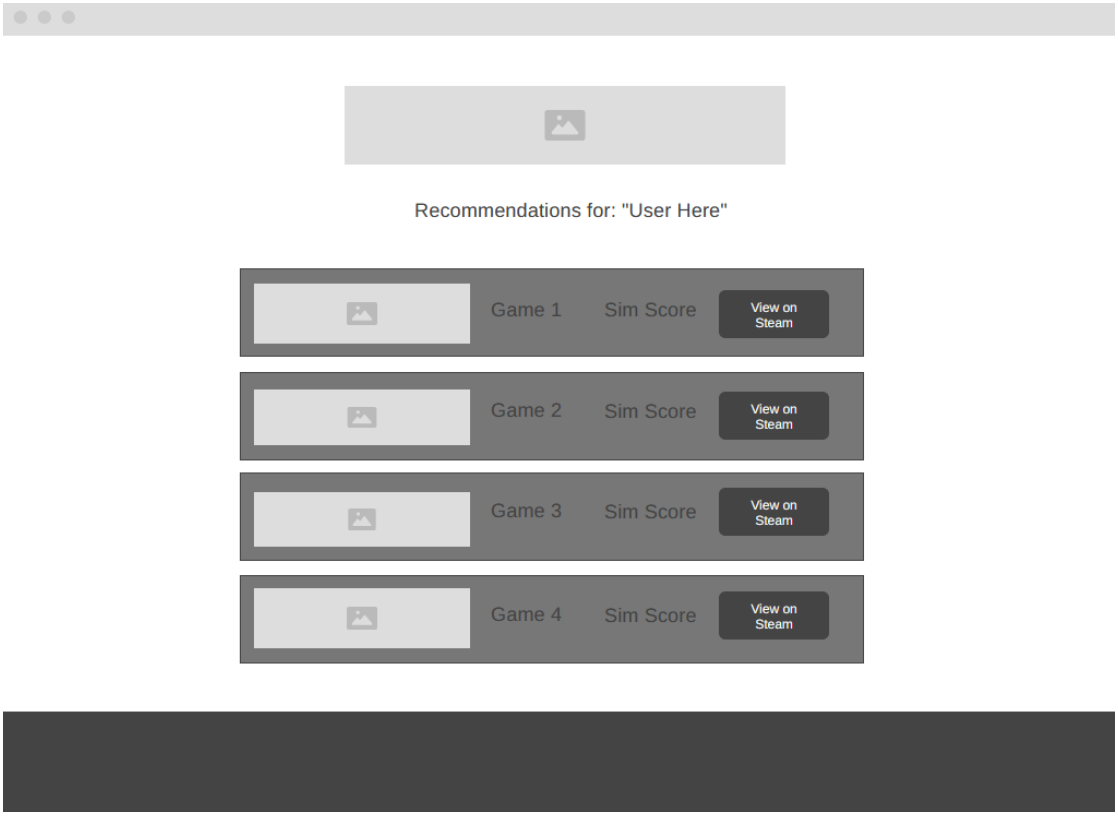

Figure 10: Recommendations Page Wireframe.

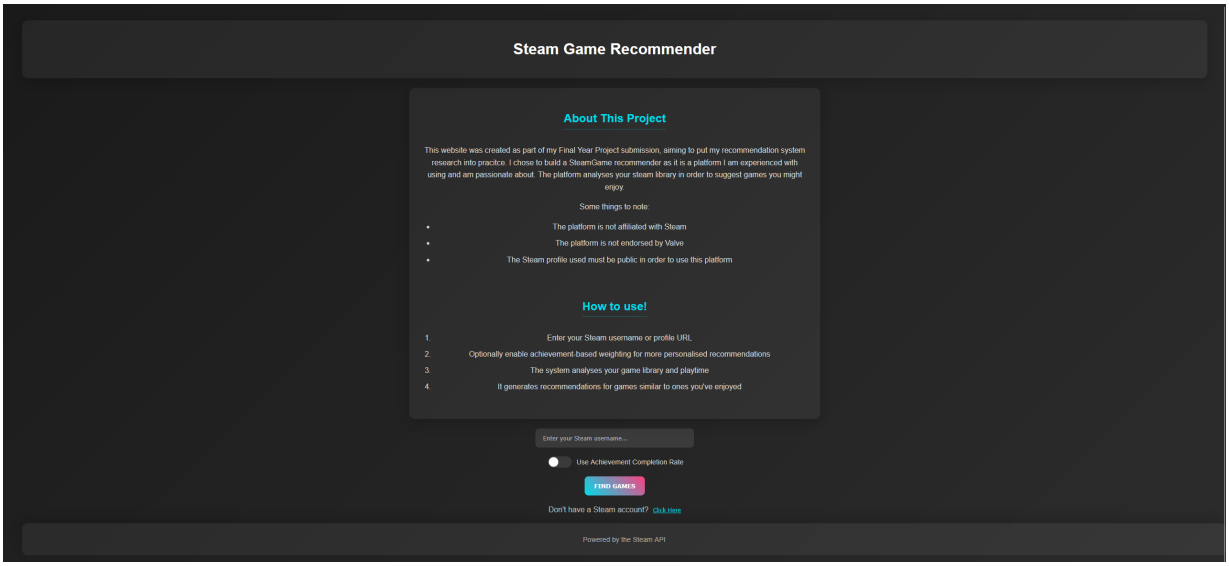

Figure 11: Landing Page Screenshot.

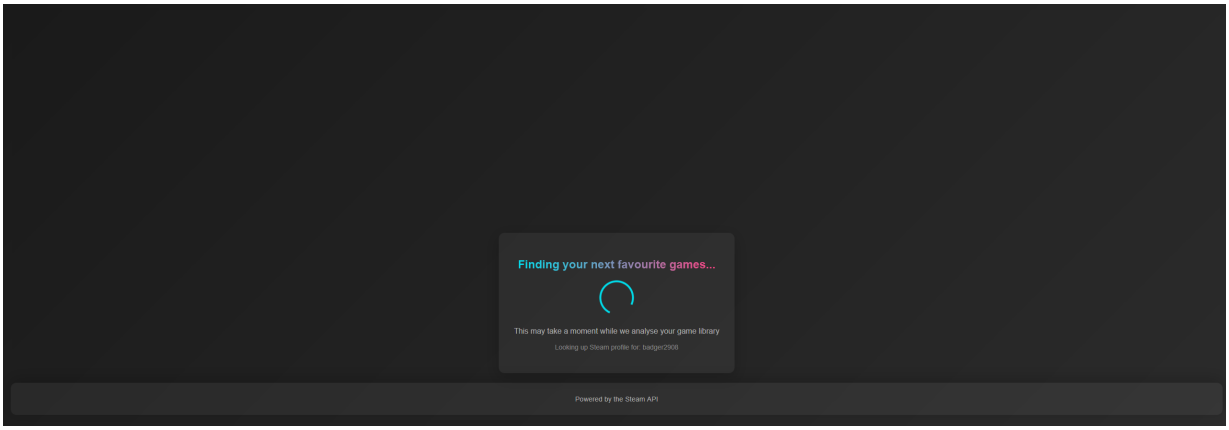

Figure 12: Loading Page Screenshot.

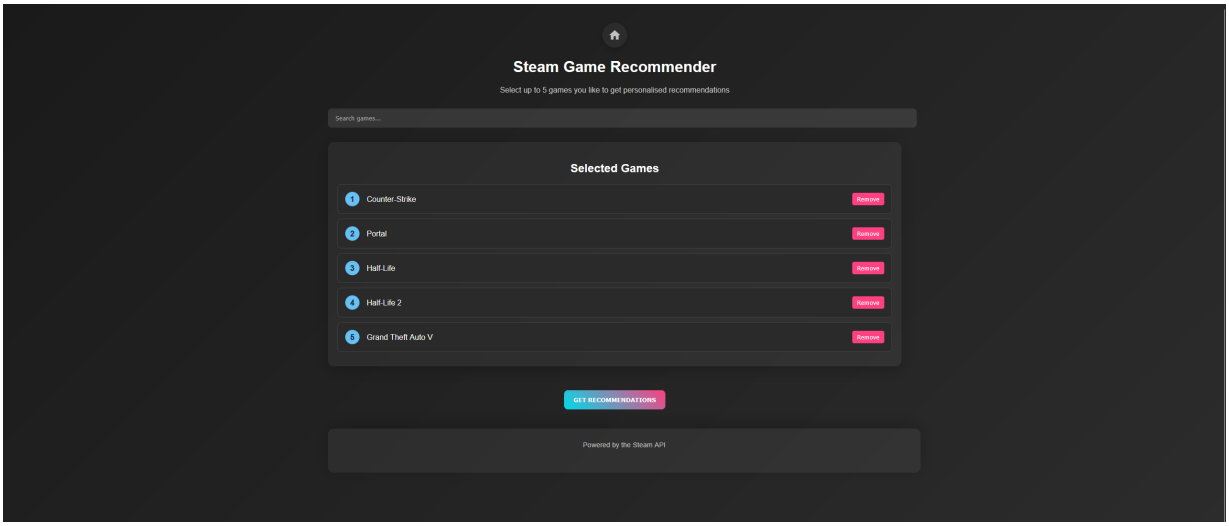

Figure 13: 'No-Steam' Page Screenshot.

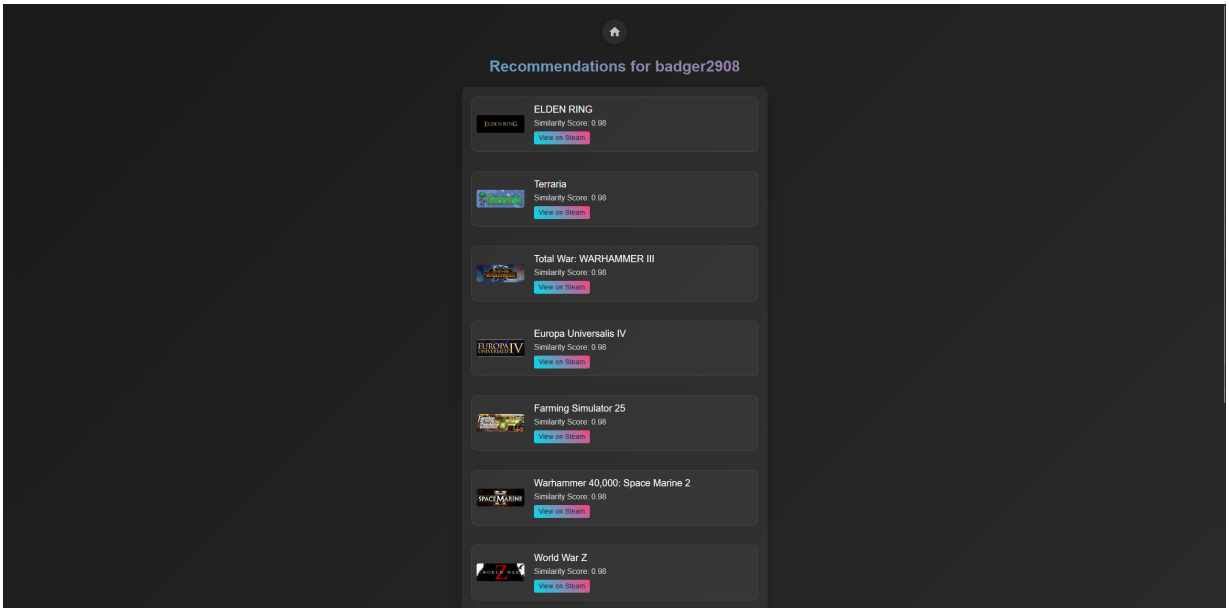

Figure 14: Recommendations Page Screenshot.

B. Code Snippets

```
def store_game_data():
    for game in games:
        appid = game["appid"]
        name = game["name"]
        genres = game["genres"]
        categories = game["categories"]
        positive_review_ratio = game["positive_review_ratio"]
        active_players = game["active_players"]
        # One-hot encode genres & categories
        genre_vector = np.array([1 if g in genres else 0
                                 for g in unique_genres])
        category_vector = np.array([1 if c in categories else 0
                                    for c in unique_categories])
        # Normalise review score and log scale active players
        review_vector = np.array([positive_review_ratio])
        active_players_vector =
            np.array([np.log1p(active_players)])
        # Create final vector
        final_vector = np.concatenate((genre_vector,
            category_vector, review_vector, active_players_vector))
```

Figure 15: Function to create the vector representation of a certain appid.

```python
def get_player_data():
    username = request.args.get('username')
    if not username:
        return jsonify({"error": "Username is required"}), 400
    try:
        # Attempt to resolve the username to a SteamID
        steam_id = username
        vanity_response = requests.get(
            f"https://api.steampowered.com/ISteamUser/
            ResolveVanityURL/v1/?key={API_KEY}&
            vanityurl={username}"
        )
        vanity_data = vanity_response.json().get("response", {})
        if vanity_data.get("success") == 1:
            steam_id = vanity_data.get("steamid")
        # Fetch owned games
        games_response = requests.get(
            f"https://api.steampowered.com/IPlayerService/
            GetOwnedGames/v1/?key={API_KEY}
            &steamid={steam_id}&format=json
            &include_appinfo=true"
        )
        games_data = games_response.json().get("response",
            {}).get("games", [])
        if not games_data:
            return jsonify({"error": "No games found for this
                user."}), 404
        # Create a dictionary of {appid: playtime}
        user_games = {int(game["appid"]):
            game["playtime_forever"] for game in games_data}
        return jsonify({"steam_id": steam_id, "games":
            user_games})
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Figure 16: Function to retrieve player Steam data, including Steam ID and owned games with playtime, using the Steam API.

```python
def get_games():

    try:
        conn = connect_db()
        cur = conn.cursor()
        cur.execute("SELECT appid, name FROM games ORDER BY name")
        rows = cur.fetchall()
        cur.close()
        conn.close()

        games = [{"id": appid, "name": name} for appid, name in
            rows]
        return jsonify({"games": games})
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Figure 17: Function to retrieve all games from the database, returning a JSON object with game IDs and names.

```python
def build_user_profile(user_games, steam_id, use_achievements=False):
    game_vectors = get_game_vectors()
    user_vector = np.zeros(num_features, dtype=np.float32)
    total_playtime = sum(user_games.values())
    if total_playtime == 0:
        print("User has no playtime data. Returning zero vector.")
        return user_vector
    # Pre-fetch achievement data if achievements are enabled
    achievement_data = {}
    if use_achievements:
        print("Fetching achievement data for all games...")
        for appid in matched_games:
            achievement_data[appid] = get_game_achievements(appid, steam_id)
        print(f"Fetched achievement data for {len(achievement_data)} games")
    for appid in matched_games:
        playtime = user_games[appid]
        weight = playtime / total_playtime  # Base weight from playtime
        user_vector += game_vectors[appid] * weight  # Weighted sum of vectors
    user_vector = normalize(user_vector.reshape(1, -1))[0]
    return user_vector
```

Figure 18: Function to create a user profile vector based on game ownership and playtime.

```python
def build_user_profile(user_games, steam_id,
    use_achievements=False):
    ..
    ..
    # Pre-fetch achievement data for all games if achievements
        are enabled
    achievement_data = {}
    if use_achievements:
        print("Fetching achievement data for all games...")
        for appid in matched_games:
            achievement_data[appid] =
                get_game_achievements(appid, steam_id)
        print(f"Fetched achievement data for
            {len(achievement_data)} games")

    for appid in matched_games:
        playtime = user_games[appid]
        weight = playtime / total_playtime  # Base weight from
            playtime
        # When achievements are enabled, adjust the weight based
            on completion rate
        if use_achievements and appid in achievement_data:
            total_achievements, completed_achievements =
                achievement_data[appid]
            if total_achievements > 0:
                completion_rate = completed_achievements /
                    total_achievements
                # Combine playtime and completion rate weights
                weight = (weight + completion_rate) / 2
                print(f"Game {appid}: Completion rate
                    {completion_rate:.2f}, Final weight
                    {weight:.2f}")
        user_vector += game_vectors[appid] * weight  # Weighted
            sum of vectors
    user_vector = normalize(user_vector.reshape(1, -1))[0]
    return user_vector
```

Figure 19: Part of the build_user_function, used to generate achievement-weighted user vectors.

```python
def no_steam_recommendations():
    try:
        data = request.get_json()
        selected_games = data.get('selectedGames', [])
        if not selected_games:
            return jsonify({'error': 'No games selected'}), 400
        game_vectors = get_game_vectors()
        # Create user vector based on selected games
        first_game_id = next(iter(game_vectors))
        vector_dim = len(game_vectors[first_game_id])
        user_vector = np.zeros(vector_dim)
        # Weight games based on their rank (higher rank = higher
            weight)
        max_rank = len(selected_games)
        for game in selected_games:
            rank = game.get('rank', max_rank)
            weight = (max_rank - rank + 1) / max_rank  #
                Normalise weight between 0 and 1
            game_id = game.get('id')
            if game_id in game_vectors:
                user_vector += game_vectors[game_id] * weight
        # Normalise the user vector
        if np.any(user_vector):
            user_vector = user_vector /
                np.linalg.norm(user_vector)
```

Figure 20: Function used on the 'No Steam' page to generate game recommendations by creating a weighted user vector from the given games.

```
test('submits form with username and achievement toggle enabled',
    () => {
    render(
      <BrowserRouter>
        <LandingPage />
      </BrowserRouter>
    );
    // Enter username
    const usernameInput = screen.getByPlaceholderText('Enter your
        Steam username...');
    fireEvent.change(usernameInput, { target: { value: 'testuser'
        } });
    // Enable achievement toggle
    const achievementToggle = screen.getByRole('checkbox');
    fireEvent.click(achievementToggle);
    // Submit form
    const submitButton = screen.getByRole('button', { name: 'Find
        Games' });
    fireEvent.click(submitButton);
    // Check navigation
    expect(mockNavigate).toHaveBeenCalledWith('
    /loading/testuser?use_achievements=true');
});
test('navigates to no-steam page when "Click Here" button is
    clicked', () => {
    render(
      <BrowserRouter>
        <LandingPage />
      </BrowserRouter>
    );
    // Click "No Steam" link
    const noSteamButton = screen.getByRole('button', { name:
        'Click Here' });
    fireEvent.click(noSteamButton);
    // Check navigation
    expect(mockNavigate).toHaveBeenCalledWith('/no-steam');
});
```

Figure 21: Snippet of the JavaScript Jest tests for the LandingPage component, verifying form submission with achievement toggle and navigation to the no-Steam page.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response structure is valid", function () {
    const responseData = pm.response.json();
    pm.expect(responseData).to.be.an('object');
    pm.expect(responseData.response).to.exist;
});

const response = pm.response.json().response;

if (response.success === 1) {
    pm.test("Successfully resolved vanity URL", function () {
        pm.expect(response.steamid).to.exist;
        pm.expect(response.steamid).to.be.a('string');
        pm.expect(response.steamid).to.match(/^\d+$/);

        // Store Steam ID for future tests
        pm.environment.set('steamid', response.steamid);
    });
} else {
    pm.test("Failed to resolve vanity URL with proper error",
        function () {
        pm.expect(response.success).to.equal(42);
        pm.expect(response.message).to.exist;
    });
}
```

Figure 22: Example of part of the POSTman test script verifying the Steam API ResolveVanityURL endpoint, checking status code, response structure, and Steam ID resolution.

C. Test Results

ID	Test Case	Steps	Expected Result
F01	Landing Page Rendering	1. Render the LandingPage component	Header, input field, achievement toggle, and submit button are displayed correctly
F02	Form Validation	1. Submit form with empty username	Form submission is prevented
F03	Achievement Toggle	1. Click achievement toggle 2. Submit form	Toggle state changes Navigate to loading page with use_achievements=true
F04	No Steam Navigation	1. Click "Click Here" button	Navigation to the NoSteamPage component
F08	Request Timeout	1. Mock API timeout 2. Render LoadingPage for 60+ seconds	Error message displayed with retry button
F15	Get Recommendations	1. Select 2–3 games 2. Click Get Recommendations button	API called with selected games, recommendations displayed

Table 13: Detailed Frontend Component Test Cases

Component	Test Description	Expected Result	Status
LandingPage	Render landing page with all required elements	All UI elements visible	Pass
LandingPage	Submit form with username only	Navigate to loading page with achievements=false	Pass
LandingPage	Submit form with username and achievement toggle	Navigate to loading page with achievements=true	Pass

Continued on next page

Table 14 – Continued from previous page

Component	Test Description	Expected Result	Status
LandingPage	Click "No Steam" button	Navigate to no-steam page	Pass
LoadingPage	Render loading state	Loading spinner and text visible	Pass
LoadingPage	Successful API response	Navigate to recommendations page	Pass
LoadingPage	API failure	Set error in localStorage and navigate to recommendations	Pass
LoadingPage	Request timeout	Show timeout error and retry button	Pass
RecommendationsPage	No recommendations in localStorage	Show "No recommendations found" message	Pass
RecommendationsPage	Valid recommendations in localStorage	Display game cards with details	Pass
RecommendationsPage	Click home button	Navigate to landing page	Pass
NoSteamPage	Initial page load	Show search input and empty selection	Pass
NoSteamPage	Search for games	Display matching game results	Pass
NoSteamPage	Select and remove games	Update selected games list correctly	Pass

Continued on next page

Table 14 – Continued from previous page

Component	Test Description	Expected Result	Status
NoSteamPage	Get recommendations	Display recommended games	Pass
NoSteamPage	Click home button	Navigate to landing page	Pass

Table 14: Frontend Component Unit Tests

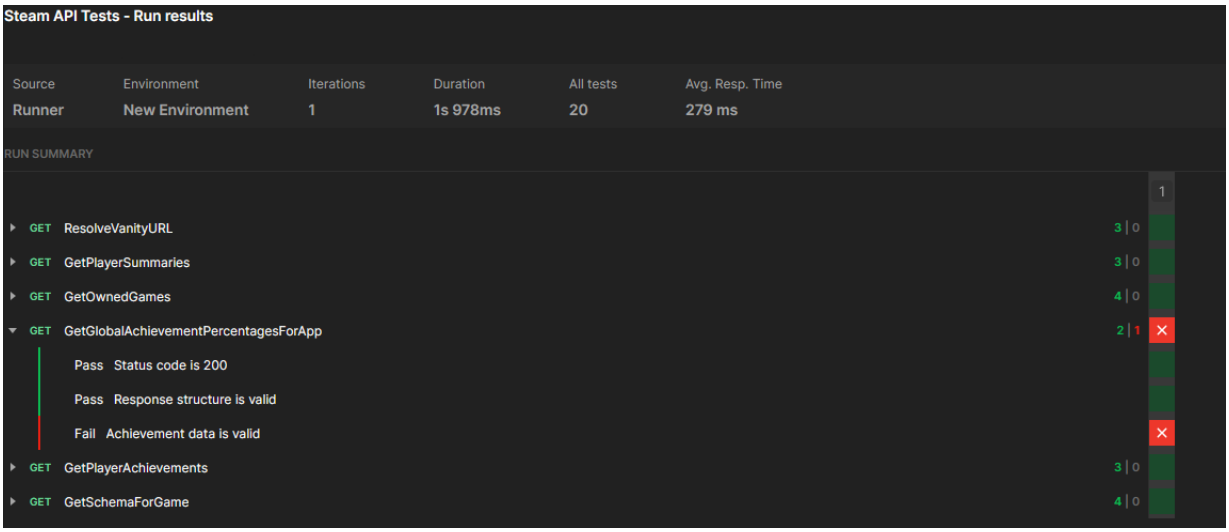

Figure 23: Screenshot of the results of the Steam API POSTman tests.

Endpoint	**Test Case**	**Validation Criteria**	**Status**
GET /player-data	Valid username	Returns Steam ID and games object	Pass
GET /player-data	Invalid username	Returns 404 error	Pass
GET /player-data	Missing username	Returns 400 error with message	Pass
GET /recommendations	Valid username	Returns array of recommendations	Pass
GET /recommendations	With achievements=true	Returns recommendations weighted by achievements	Pass
GET /recommendations	Invalid username	Returns appropriate error	Pass
GET /games	Basic request	Returns array of games with IDs and names	Pass
POST /no-steam-recommendations	Valid selected games	Returns array of recommendations	Pass
POST /no-steam-recommendations	Empty selection	Returns 400 error	Pass

Table 15: Backend API Endpoint Tests