Movie Recommender Project Document

Group 5: Rachel Kelly, team member 2, team member 3, team member 4, team member 5

INTRODUCTION

Aims and Objectives

The primary objective of the project is to build a cross-platform movie recommendation system. The recommendations are based on the user's top 5 movies and their responses to a magazine-style quiz, which group users with specific “vibes”. By providing customised content, the system seeks to increase user interaction and satisfaction.

Report Roadmap

This document is organised into the following sections:

- Introduction: Outlines the aims and objectives of the project.
- Background: Provides context on the project and the problem it solves.
- Specifications and Design: Presents the functional and non-functional requirements, design and architecture.
- Implementation and Execution: Describes the development approach, team member roles, tools and libraries used, implementation process and challenges.
- Testing and Evaluation: Details testing strategy and system limitations.
- Conclusion: Summarises project outcomes and potential future work.

BACKGROUND

Project Overview

The team pooled ideas for possible projects, shortlisted a set of candidate projects, reviewed the feasibility of implementing each within the assignment timescale using skills available within the team, and decided to choose the Movie Recommender. The Movie Recommender is designed to solve the problem of providing users with personalised movie suggestions.

Recommendation systems are fundamental to platforms such as Netflix and Spotify which rely on algorithms that analyse big data on users’ consumption patterns. The team identified that, although each platform offered recommendations, there was a gap in offering cross-platform recommendations tailored to a particular user’s interests. Initially, the team considered recording data about movies seen and users’ interests and using those data to generate recommendations but, without auto-data collection, this placed a data entry burden on users, for recording and potentially rating movies they had seen, and required potentially complex management of a “folksonomy” of users’ interests to find similar users. The team decided instead to ask each user to enter their top 5 movies and complete a simple, magazine-style quiz that would allow similar users to be grouped so that the most popular movies from similar users could form the basis of recommendations broadened by search and recommendation features available through the TMDB API.

The quiz presents 5 multiple choice questions to group users with similar vibes, for example:

Your partner moves across the world…what do you do?
A: Move with them! They're the love of your life and you're always up for an adventure!
B: You'll do long distance and see how it goes. You could never move away from home!
C: Call it quits. You've got your own plans and you're not changing them for anyone!

★ Mostly A: Hopeless romantic
★ Mostly B: Comedy-loving homebody
★ Mostly C: Independent thrill-seeker

The solution uses a combination of the user's top 5 movies and their quiz-assigned vibe to generate a list of 25 movie recommendations ranked by popularity.

SPECIFICATIONS AND DESIGN

The following functional and non-functional requirements were identified to deliver the solution.

Functional Requirements

- New user registration
- Existing user log-in
- Input and storage of top 5 movies
- Quiz functionality with categorised vibes
- Generation of movie recommendations

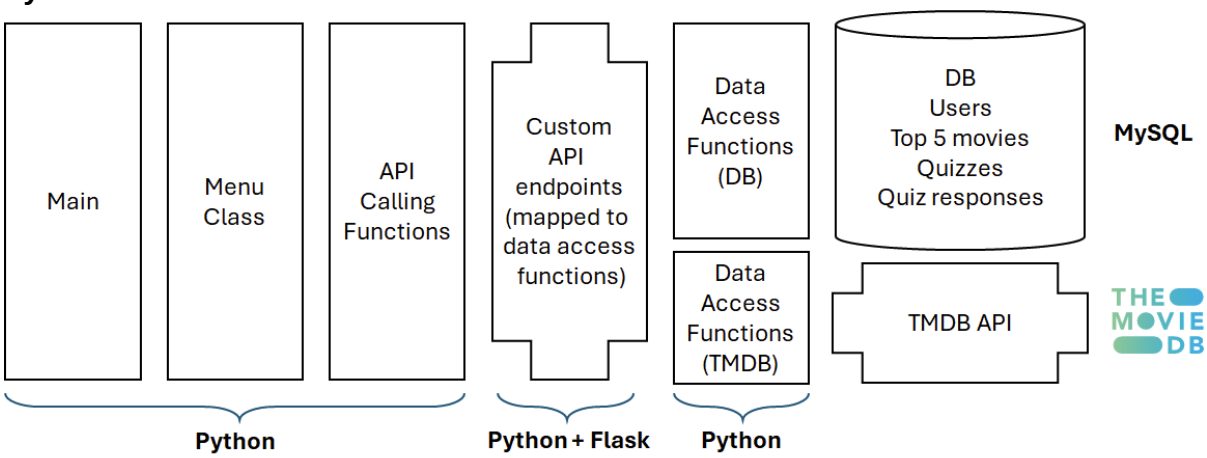
Non-Functional Requirements

- Scalability – separate back end and front end to enable scaling
- Maintainability – follow separation of concerns and SOLID principles with the solution decomposed into small, well-named components that perform clearly identifiable and testable functions
- Supportability – adopt industry-standard technologies and solutions
- Security – avoid vulnerabilities such as SQL injection
- Intuitiveness – use clear messages so the user knows what to do at each stage

Design and Architecture

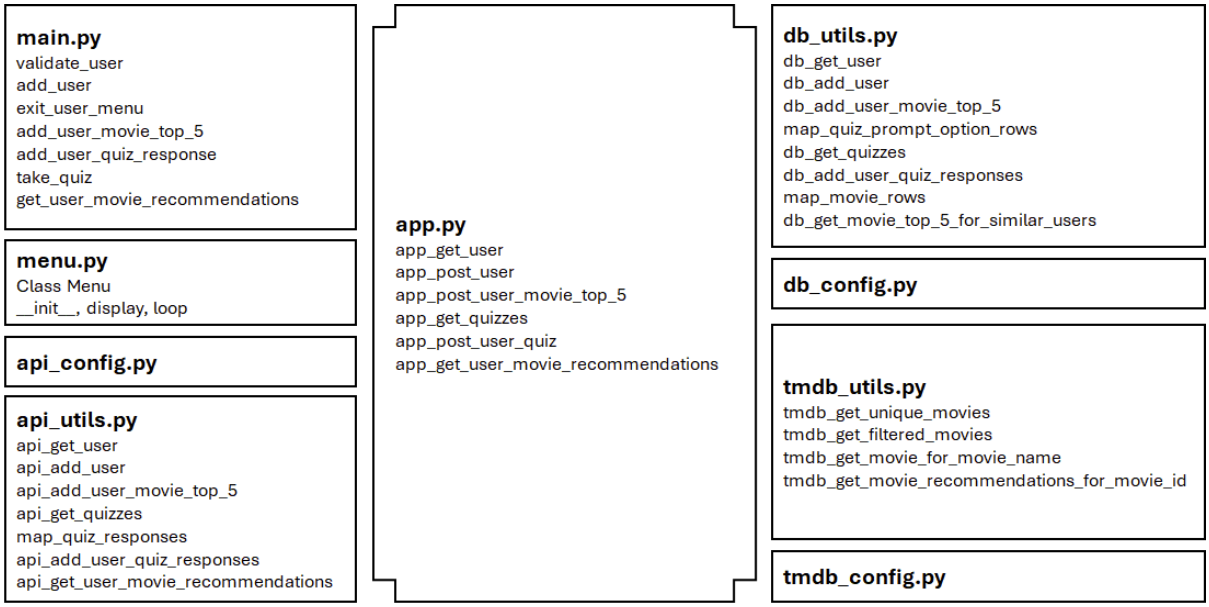
To satisfy these requirements, the team designed a tiered architecture in which back-end data sources were accessed through utility helper functions and exposed via a set of custom API endpoints consumed by a front-end user interface using a Menu class and API helper functions.

System Architecture



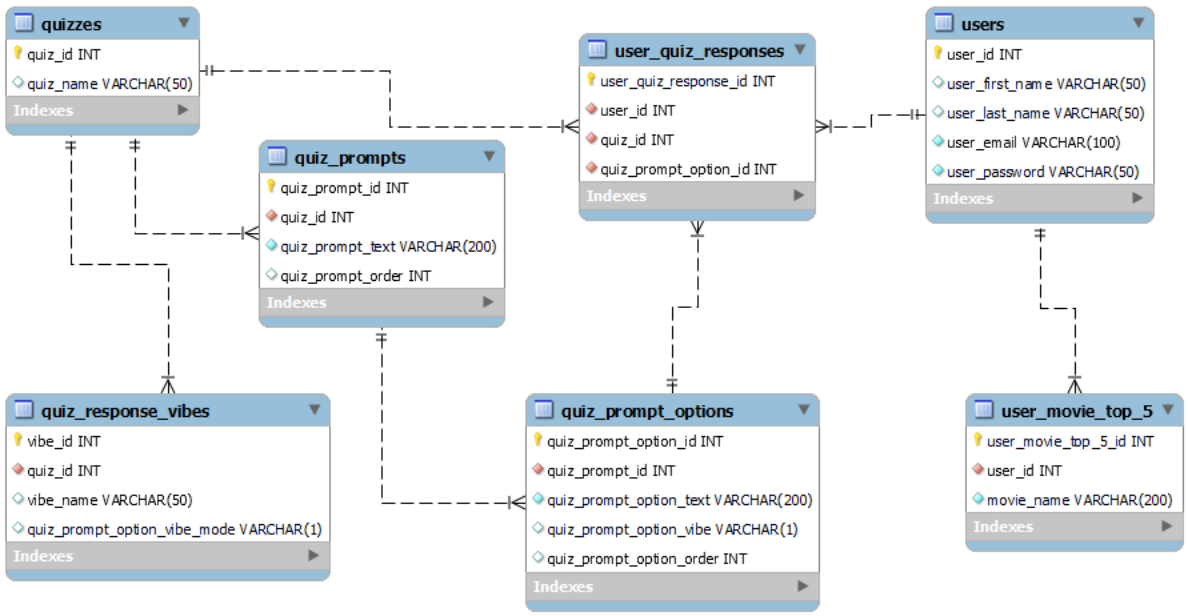
Python Architecture

The Python component is structured into distinct modules, with functionality as follows:



Database Architecture

A relational database model holds data about users, their top 5 movies, quizzes and quiz responses, with appropriate key constraints to ensure referential integrity. Views present the results of complex joins required to retrieve movies for similar users, and stored procedures can be called to add users, their top 5 movies and their quiz responses.



IMPLEMENTATION AND EXECUTION

Development Approach and Roles

The project was developed using an agile approach, organised into weekly sprints with work coordinated through Jira and regular Slack communication and team meetings. The team adopted specific roles:

- **Product Owner (PO)**: Rachel Kelly - Responsible for defining the vision, prioritising the backlog and ensuring the final product met user needs.
- **Scrum Master (SM):** team member 2 - Responsible for facilitating the agile process, writing Jira tickets, coordinating work and removing obstacles.
- **Development Team (DT)**: team member 3, team member 4, team member 5- Responsible for implementing features, managing the codebase and ensuring quality.

Implementation Process

Feature priorities were agreed with the PO and the SM created a backlog of tasks to complete by prototyping aspects of the solution and creating Jira tickets for each deliverable. After researching best practice, tickets were structured as follows:

- User Story - “As a user, I want to …, So that I can …”
- Acceptance Criteria - list of precise, testable statements
- Technical Considerations
- Design

In total, 75 tickets were written and organised into sprints as follows:

Sprint		Tickets	main.py	menu.py	api_utils.py + config	app.py	tmdb_utils.py + config	db_utils.py + config	user_movie_vibes.sql	Test suite	Chores	Parent task
1 (17)	Add feature for adding user	10	1		2	1		3	2			1
	Add feature for validating user	5	1		1	1		1				1
	Add generic feature for displaying menus	1		1								
	Add feature for displaying user menu	1	1									
2 (29)	Add feature for adding user's top 5 movies	9	1		1	1		1	3		1	1
	Add feature for presenting and storing quiz responses	20	2		3	2		3	8		1	1
3 (34)	Add feature for presenting list of movie recommendations	13	1		1	1	4	2	3			1
	Add feature for displaying main menu	1	1									
	Add enhancement for validating user input	1	1									
	Add enhancement for handling API exceptions	1			1							
	Add test suite for all functions and classes	7								6		1
	Perform end-to-end interaction testing	1									1	
	Add README detailing code deployment	1									1	
	Write Project Document	1									1	
	Standardise and comment all code	1									1	
	Complete Project Activity log	1									1	
4 (1)	Create presentation	1									1	
		75	9	1	9	6	4	10	16	6	8	6

Each sprint had specific goals and deliverables, allowing the team to iteratively develop and improve the Movie Recommender system. The focus of each sprint was as follows:

Sprint	From	To	Focus
Sprint 1	2024-08-05	2024-05-12	Establish system foundations and deliver features for user registration, login and menus
Sprint 2	2024-08-12	2024-08-19	Add features for adding top 5 movies and taking a quiz
Sprint 3	2024-08-19	2024-08-25	Add feature for movie recommendations; improve, test, standardise and document solution

Tools and Libraries

In delivering the solution, the team used the following tools and libraries:

- **Integrated Development Environment:** PyCharm, MySQL Workbench
- **Python standard libraries:** itertools, json, re, unittest
- **Python third-party libraries:** mysql-connector-python, requests, python-dotenv
- **Python frameworks:** Flask for API development
- **Database:** MySQL for data storage and retrieval
- **External APIs:** TMDB API for fetching movie data
- **Version control:** GitHub for managing code versions
- **Project management:** Jira for task management and sprint planning

For the final sprint, the team concentrated on polishing the system, enhancing existing features and delivering comprehensive testing and documentation.

- **Enhancements:** Added input validation to ensure data integrity and error handling for API exceptions to enhance system robustness.
- **Testing:** Developed a comprehensive test suite for all functions and classes and conducted end-to-end user interaction testing to ensure a seamless user experience.
- **Documentation:** Standardised and commented all files to improve readability and maintainability and added a README.md file detailing deployment.

Throughout each sprint, the team used agile practices, including stand-ups, sprint planning meetings and retrospectives. Alongside daily communication, these practices ensured focus and continual improvement of the development process. The team worked with a shared repository on GitHub with branches for each team member. A protocol was agreed for referencing Jira ticket(s) in Pull Request (PR) titles and including other team members as reviewers to engage the whole team in code reviews.

Implementation Challenges and Solutions

The team had to juggle the project with work commitments, and the retrospective at the end of Sprint 1 revealed a day's slippage. Sprint 2 involved more complex code which was delivered alongside more challenging outside commitments. Team members supported one another in resolving blockers and an iterative approach was adopted whereby Acceptance Criteria for Jira tickets were refined to promote good practice identified in code reviews.

In response to slippage in Sprint 2 and the complexity of dependencies, the SM allocated Sprint 3 tickets into two-day slots, so that team members could pull the latest version of the

codebase at the start of the slot and create their PR at the end, without compromising the work of others.

Personal environment configurations proved challenging to maintain when pulling from and pushing to a shared repository so the SM proposed storing personal configurations (such as the bearer token for the external API and MySQL credentials) in an .env file specified in .gitignore to ensure they were excluded from pushes.

Although tickets were not of equal complexity, the number of tickets completed in each sprint demonstrated increased velocity as foundations were laid, environment issues were resolved, and more code became available for developers to learn from.

Throughout the implementation process, Slack communication enabled rapid resolution of blockers and Zoom meetings provided an important way to coordinate efforts, maintain morale and celebrate successes.

TESTING AND EVALUATION

Testing Strategy

Testing was an integral part of the team's approach throughout the project:

- Sprints were organised to build from the back end outwards and successively add additional features so there was always a working solution to test.
- Jira tickets were specified with precise Acceptance Criteria so that developers could confirm their code met expectations at each stage.
- API endpoints were tested with Insomnia.
- In Sprint 3, the complete set of Acceptance Criteria was used to develop a test suite with unit tests for every function and class in each Python file.

Components	Test Cases	Status
db_utils.py	10	Passed
tmdb_utils.py	4	Passed
app.py	10	Passed
api_utils.py	13	Passed
main.py	17	Passed
menu.py	8	Passed

Functional and User Testing

- **Unit tests** were developed for individual functions and modules to ensure their correct operation in isolation. Unit tests used decorators to mock functions and components such as Flask requests and DB connections and assert tests to undertake black box testing – confirming that input data produced expected outputs, either as return values or display output – and white box testing – confirming correct passing of parameters to called functions. Test cases were written for success and failure paths.
- **Integration tests** verified the seamless interaction between different components of the system, including the database, API and user interface.
- **User testing** involved the entire team in evaluating the system's usability, intuitiveness and effectiveness in providing movie recommendations.

Testing Challenges and System Limitations

Testing functions called from the Menu class initially posed a challenge due to its recursive nature; however, these challenges were overcome by mocking user input to ensure that options chosen did not cause test menu instances to loop indefinitely.

As the system was only deployed to team members' laptops, rather than a server capable of hosting Python and the back-end DB, it was not possible to perform meaningful performance testing or comprehensive testing of concurrent access. This would have required specialist user input simulation and load testing software. Additionally, the system depends on the TMDB API for movie data, which is being accessed under a free-for-education arrangement, which imposes throttling and rate limits that would prevent meaningful performance testing.

CONCLUSION

The Movie Recommender system successfully achieved its objective of providing personalised movie recommendations based on a user's top 5 movies and quiz responses, and met all the Must-Have and Nice-to-Have requirements of the brief:

MUST HAVE	
Clearly-defined objective	Providing cross-platform movie recommendations to users based on their top 5 movies and responses to a magazine-style quiz
Existing API	TMDB API to access movie details and provide recommendations
OOP principles and libraries	Object-oriented Menu class with different instances for each menu; itertools to filter and get unique lists of movie recommendations

NICE TO HAVE	
Python – API – DB interaction	DB of users, top 5 movies, quizzes and responses, maintained via custom API endpoints
Own API endpoints	To manage data about users, top 5 movies, quizzes and responses in the DB and get recommendations
Decorators	To map functions to own endpoints and menu options, and to test using mocked functions
Recursive functions	To display options within the Menu class

The project demonstrated effective use of Python, Flask, MySQL and external APIs to create a scalable and user-friendly application delivered through an agile development process that made extensive use of Jira. Future enhancements could include expanding the quiz options, improving the recommendation algorithm and integrating additional data sources for more tailored recommendations.