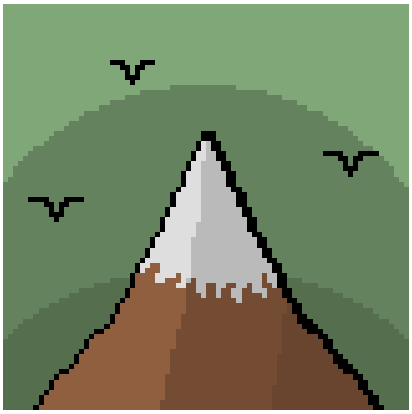
Peak Adventure Experiences



Software Testing Plan

Date: 03/20/2026
Team Members: Alonso Garcia, Jack Morris, Yahir Espinoza, Makaela Crookes
Project Sponsor: Paddy McGarry
Team Mentor: Ogonna Eli

TABLE OF CONTENTS

I. INTRODUCTION

Relocating to a new city is a complex and high-impact decision, requiring individuals and families to evaluate financial, social, and lifestyle factors. This project presents a web-based virtual relocation experience for the city of Flagstaff, designed to help prospective homeowners and relocators better understand daily life through an interactive, game-based environment. By combining exploration, non-player character (NPC) interactions, and mini-games, the system provides a more immersive alternative to traditional relocation tools. The primary quality goals of the application are usability, ensuring users can quickly learn and navigate the system; performance, maintaining smooth gameplay and responsive system interactions; and reliability, ensuring consistent access to features such as user data, progress tracking, and external information like weather updates.

Software testing for this project is conducted within the boundaries of a web-based architecture consisting of a Next.js frontend, a Phaser-based game engine, a FastAPI backend, and a PostgreSQL database. The system also integrates external services, such as real-time weather APIs. Testing focuses on core system components, including user authentication, gameplay mechanics, frontend-backend communication, and database interactions. These components are critical to ensuring a functional and engaging user experience. Out of scope for testing are third-party service reliability, large-scale production deployment beyond the requirement of supporting at least ten concurrent users, and compatibility with outdated or unsupported browsers. Testing will be performed in local development and browser-based environments, with validation aligned to project milestones and demonstration deliverables outlined in the development backlog and demo flight plan.

The testing strategy for this system incorporates multiple levels of validation throughout the development lifecycle. Unit testing will be used to verify the correctness of backend logic and API endpoints, and will be executed during development and through continuous integration on each code update. Integration testing will ensure proper communication between the frontend, backend, and database, and will be conducted prior to major demonstrations and milestone reviews. Usability testing will evaluate how effectively users can navigate the application, interact with NPCs, and complete objectives, with sessions planned before key milestones such as Alpha II and the acceptance demonstration. Performance testing will validate that system requirements are met, including maintaining frame rates above 30 FPS and ensuring that data such as weather updates and map interactions respond within specified time constraints. Security testing will focus on authentication and data protection, ensuring that user information is handled securely and in accordance with project requirements.

This testing strategy prioritizes areas of highest risk and user impact. Usability and performance are emphasized due to their direct influence on user engagement and the overall effectiveness of the virtual relocation experience. Because the application relies heavily on

interactive gameplay, poor responsiveness or unintuitive design would significantly reduce its value. Additionally, authentication and data handling require careful validation to ensure reliability and maintain user trust. In contrast, less emphasis is placed on static content and third-party integrations, as these pose lower risk within the system's scope.

The following sections provide a more detailed description of each testing method and how it is applied within the system.

II. UNIT TESTING

Unit testing for the Virtual Flagstaff project is a way for us to test the project's functionality by testing individual modules of the project to ensure functionality and correctness. The back-end's and database's tests will be performed using Python's built in unittest function along with the pytest library for additional testing and test assurance. These tests should be mostly automated thanks to how robust both the unittest function and pytest library are in their functionality. Testing metrics here would include the coverage and pass/fail metrics, coverage to help inform about the percentage of code that's been tested and pass/fail to indicate where and how our backend can fail.

The front-end React webpage can be tested, however, due to Phaser's heavy reliance on both a 'god-object' and rendering functionality, the bulk of this project, the Phaser gamified experience, cannot be unit tested without immense effort put into creating mock objects and functions. The React part of our project's front-end will be tested using the built in React Testing Library and Jest, a Javascript library made for unit testing. These tests will likely be made to be run locally for when issues occur when using the React webpage. In this case the only metric that would often be important would be pass/fail to find potential edge cases where our webpages can crash or otherwise fail.

The units that should be tested are those from the back-end that do important computations or that communicate with other parts of the system, such as the log-in, register, session fetch, token creation, and weather functions. Additionally, the front-end related portions of these systems should also be tested, such as the login page and the register page.

In order to make proper and functional tests, we will need to determine what cases to test and what those cases' outputs should be. To do so we will logically evaluate what each function is intended to do and determine what is and is not a valid output from there. Then, we'll create test cases that easily fit within these circumstances to get the expected output and create a few more test cases that push the bounds of what should happen and see what occurs from them, and use those results to fix functionality as needed. A few example tests have been outlined below.

II.I UNIT TEST DESIGN APPROACH

Unit Under Test: register_user(self, user, dbSession, response)
Purpose: Registers the current user with the database
Test Case Categories:

- Valid inputs: valid user, session, and response
- Boundary cases: very long username or identifier
- Invalid inputs: no user, null session, or null response

Sample Tests:

- Register with an 8 character username and valid identifier
- Register with a very long username and a valid email
- Register with an 8 character username and a very long email
- Register with no user

Unit Under Test: login_user(self, response, credentials, session)
Purpose: Logs-in the user with their credentials
Test Case Categories:

- Valid inputs: valid credentials, session, and response
- Invalid inputs: invalid credentials, null session, or null response

Sample Tests:

- Log-in with valid credentials
- Log-in with valid credentials but no session
- Log-in with invalid credentials

Unit Under Test: get_current_user(token, session)
Purpose: Retrieve the user data from their token and session to prevent repetitive logins
Test Case Categories:

- Valid inputs: valid token and session
- Invalid inputs: invalid token or invalid session.

Sample Tests:

- Fetch with valid token and session
- Fetch with valid token but invalid session
- Fetch with invalid token but valid session

Unit Under Test: create_access_token(data, expires_delta)
Purpose: Create an access token for the user to reduce how often they need to sign in.
Test Case Categories:

- Valid inputs: valid data and proper expiration
- Border inputs: valid data but improper or no expiration
- Invalid inputs: invalid data

Sample Tests:

- Create a token with valid data and expiration date
- Create a token with valid data but very long expiration date
- Create a token with valid data and no expiration date
- Create a token with invalid data but very long expiration date

Unit Under Test: get_forecast(self, period)
Purpose: Get the forecast from the weather api
Test Case Categories:

- Valid inputs: current (near future) period
- Border inputs: far future or past period
- Invalid inputs: no period

Sample Tests:

- Fetch the forecast set for one hour
- Fetch the forecast set for one day
- Fetch the forecast set for one month
- Fetch the forecast set for one year
- Fetch the forecast set for one day ago
- Fetch the forecast set for one year ago

III. INTEGRATION TESTING

Integration testing for this project verifies that independently developed components function correctly when combined. Following Alpha I, the system includes multiple interacting components, including the frontend web application, backend API services, database, and external APIs. While individual components may function correctly in isolation, integration testing ensures that data flows, system contracts, and communication between components operate reliably under real-world conditions. Particular focus is placed on identifying failures at system boundaries, such as data mismatches, API contract violations, and configuration issues that could disrupt core user workflows.

The primary integration points in this system include interactions between the frontend and backend API, backend services and the PostgreSQL database, authentication and authorization flows, and communication with external services such as the weather API. These integration points are critical because failures in these areas directly impact user experience, including account access, gameplay functionality, and real-time data display.

Integration tests will be conducted in a controlled development environment using a locally hosted backend and database, along with browser-based frontend testing. A dedicated test database will be used to isolate test data from production data. Test user accounts and predefined datasets will be created to simulate realistic user interactions. Test data will be reset or reused between test runs to ensure consistency and reproducibility. External API calls, such as weather data requests, will be tested using live or mocked responses depending on availability and reliability. This environment ensures that integration issues can be consistently reproduced and diagnosed.

III.I. INTEGRATION SCENARIOS

Integration Point: Frontend – Backend API – Weather API
Feature: Real-time Flagstaff, Arizona weather
Scenario Description: Real-time weather Flagstaff, Arizona weather is displayed on web interface
Integration Steps:

- Frontend sends a GET request to backend API
- Backend API sends a GET request to weather API
- Weather API returns a success response with Flagstaff, Arizona weather data
- Backend API returns a success response with weather data
- Frontend stores weather data in web storage to avoid making future requests
- Frontend displays weather data in landing page or game page

Expected Results:

- API request is accepted
- Weather data is stored in web storage
- UI displays Flagstaff, Arizona's weather

Failure Handling:

- Backend API handles weather API errors and returns appropriate error to frontend

Integration Point: Frontend – Backend API – Database
Feature: User data acquisition
Scenario Description: A user registers an account and completes a survey, and survey is sent to client
Integration Steps:

- User submits registration form, where they want to receive texts/emails in the frontend UI
- Frontend validates user data
- Frontend sends a POST request to the backend API with user data
- Backend API validates input and stores user data in database
- Backend API validates future contact and sends survey data to client
- Backend API returns success response
- Frontend displays the user successfully registered and assigns the user a token for future account authorization

Expected Results

- API request is accepted and validated
- User's data is stored in database
- Client receives user's survey responses
- UI reflects the successful registration
- User receives a token

Failure Handling:

- Invalid input results in a clear validation error
- Database failures return appropriate error messages

Integration Point: Frontend – Backend API – Database
Feature: Display leaderboard of each game
Scenario Description: Each mini-game is displayed with a leaderboard of all users's high score
Integration Steps:

- User starts a mini-game in frontend game page
- Frontend displays base menu of the mini-game
- Frontend sends a GET request to backend API
- Backend API queries all high scores of each user for mini-game in descending order
- Backend API returns a success response with leaderboard
- Frontend updates mini-game menu with leaderboard

Expected Results

- API request is accepted
- Leaderboard is displayed on mini-game menu

Failure Handling:

- Database failures return appropriate error response

Integration Point: Frontend – Backend API – Database
Feature: NPC assigns quest to user
Scenario Description: When user talks to NPC, NPC gives user a quest
Integration Steps:

- User talks to NPC in frontend game page
- Frontend sends a POST request to backend API with user and quest data
- Backend API validates input and assigns quest to user in database
- Backend returns a success response
- Frontend updates UI to display quest for user

Expected Results

- API request is accepted and validated correctly
- User quest data is stored in database
- UI reflects successful creation without errors

Failure Handling:

- Database failures return appropriate errors
- Authentication errors prevent unauthorized access

Integration Point: Frontend – Backend API – Database
Feature: User receives rewards
Scenario Description: After user completes a quest, they are rewarded
Integration Steps:

- User completes a quest in frontend game page
- Frontend sends a PATCH request to backend api with user and quest data
- Backend API validates input and marks user's quest as completed
- Backend API queries for rewards related to quest and assigns it to user
- Backend returns success response with rewards
- Frontend displays the rewards to user in UI

Expected Results

- API request is accepted and validated correctly
- User receives rewards and can access it

Failure Handling:

- Database failures return appropriate errors
- Authentication errors prevent unauthorized access

IV. USABILITY TESTING

Usability testing for Virtual Flagstaff is centered around understanding how effectively users can navigate through and interact with the virtual environment to complete tasks, such as exploring downtown Flagstaff, interacting with NPCs to complete quests, and play minigames found throughout the map. The goal is ultimately to ensure that the system is intuitive, engaging, and accessible, allowing users to easily understand the controls laid out and explore the virtual environment with little to no hassle. In order to accomplish this, the use of usability testing will help identify any confusing interactions, validate gameplay workflows, reduce user error, and improve the overall user experience.

The target users for Virtual Flagstaff include prospective residents, students, and general users interested in exploring Flagstaff virtually before traveling. These users are expected to have varying levels of technical knowledge, so it is important for us to accommodate all users. Poor usability could lead to frustration, reduced engagement, and abandonment of the application.

The usability testing approach will consist of a combination of task based user sessions, informal feedback, and internal expert reviews. Testing will involve a handful of participants, including students and a variety of individuals who are unfamiliar with our project. Each session will have users test the game by playing through the tutorial the team has created that includes:

- Navigating through the virtual map of downtown Flagstaff
- Interacting with NPCs and completing quests
- Playing through mini-games
- Entering / Interacting with buildings and signs found throughout the map

During these sessions, the team will take note of task completion rates, user feedback, and any errors or confusion points. With the data collected from the usability testing, the team will make any necessary adjustments to improve the game and resolve any errors. Usability testing will be integrated throughout the development lifecycle as a continuous feedback loop. And by continuously incorporating user feedback, Virtual Flagstaff ensures that usability remains a central focus, resulting in an engaging and intuitive virtual exploration experience.

V. TESTING WORKFLOW & QUALITY CONTROLS

Virtual Flagstaff's defect management system will be recorded using issue tracking within the Virtual Flagstaff github repository. Each defect should include the necessary details so that other developers might be able to reproduce and correct the listed defect with only the information given in the issue.

V.I. BUG REPORT TEMPLATE

Title: A concise sentence that accurately describes the issue
Severity: A rating to convey the impact of the bug on the system (Low, Medium, High, Critical)
Priority: A rating to define how important fixing this bug is (1-4)
Bug Description: A summary of what defect is occuring
Reproduction:

1. Steps to recreate the issue
2. …

Comments: Any other relevant information for recreation
Expected Result: A summary of what should be happening if the bug did not exist
User Story: The user story the defect falls under (US-1, US-2, etc.)

V.II. SEVERITY RATINGS

Critical: A game breaking issue (Game crashes, Data loss)
High: A major functionality issue that directly impacts ability to play (Interaction button not working)
Medium: A minor functionality issue that does not impact ability to play (Unable to access pause menu UI)
Low: A minor visual/cosmetic issue that does not impact ability to play (Non-centered NPC dialogue)

V.III. PRIORITY RATINGS

1 (Immediate): Must be fixed at the soonest possible time. Recognized by a critical severity rating.
2 (High): Should be fixed as soon as time permits. Recognized by a high frequency of occurrence or a Medium-high severity rating.
3 (Normal): Fix whenever time allows. Recognized by a medium or lower severity rating and a lower frequency of occurrence.
4 (Low): Can be fixed but not necessary. Recognized as being a nice-to-have fix or polish.

V.IV. DEVELOPMENT CYCLE CRITERIA

A successful development cycle is one that has addressed all High-Critical severity defects prior to the end of that cycle. All core features should be fully and properly functioning, no new High-Critical issues have been introduced, and performance is retained.

Acceptable Issues:

- Minor UI bugs rated at medium or lower severity
- Any purely cosmetic bugs
- Any uncommon and medium or lower severity bugs

Unacceptable Issues:

- Game crashes or freezes
- Extreme decreases in performance
- Broken core features

V.V. QUALITY CONTROL

To ensure consistent quality there will be a few required quality control practices implemented. Developers are expected to run and test the game after each bug fix or feature addition and ensure it works as intended. Developers should test all aspects of the game with ample time before the end of a development cycle in order to make sure no new bugs are introduced through other features or fixes.

VI. CONCLUSION

In order for the Virtual Flagstaff project to come together and be successful, we will need to incorporate all of the above forms of testing in order to ensure project quality and usability. Unit testing will be critically important among the back-end, where visibility of what's happening is minimal, making unit testing the most efficient method of testing the back-end's functionality and determining where issues can arise there. Then, our integration testing will be used to ensure that both the front-end and back-end stay properly connected and that data is not only sent between them properly but also securely stored and retrieved when needed. Finally, our usability testing is how we will ensure that the project's front-end experience is not only functional, but that it is intuitive and simple enough for all users to use, understand, and enjoy. Using these three forms of testing as we have laid out will result in our project being as bug free and functionally usable as possible.