CEREO Living Atlas

Project Testing and Acceptance Plan

Center for Environmental Research, Education, and Outreach (CEREO)

Living Atlas Development Team

Garoutte, Zachary

Simmons, Jonathan

Gao, Yaru

March 20th, 2024

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

The CEREO Living Atlas will be tested on several improvements that will be implemented in order to achieve stable performance at a large scale and a wider array of features available to each user. Stable performance is the most crucial test for the updated application, as slowed performance during high user traffic would limit the size of the application's userbase. Users with a high level of authorization should have the ability to add new data to the map from the frontend, and to view, edit, or delete their own contributions later. Users should also have the ability to bookmark any card from the site, and to be able to easily navigate the application to access their bookmarked cards. Users should be able to sort cards by the newest cards added or by how close cards are pinned to their current location. Users should be able to reset their password using two-factor authentication via email. Map feature layers should be able to be toggled on or off. The option to upload custom thumbnails should be given when adding new cards to improve each card's design.

I.2. Test Objectives and Schedule

The objective of these tests is to ensure that the application meets our expectations set for its performance and array of features by specifically testing project requirements. In terms of performance requirements, additional software such as Lighthouse will be used to test whether the application passes the set requirements for load times. For feature requirements, we can use Jest as a testing framework and set up test cases for each feature. There may be some features that cannot be tested using a test case, and in this case the feature will need to be tested by using the feature within the application and ensuring the feature has the desired functionality.

A unit test case for a feature will be set up following the feature's implementation. The test case will be run and its output will be compared to the expected output for that feature. If the output is expected, the test will be considered passed. If the output is not expected, the problems with the output will be added as a bug report to the issues tab in the project's GitHub repository. Once this bug is considered fixed, the test will be run again, and the output will be compared to the expected output as previously. A similar testing cycle will be used with performance tests, however, performance issues may not be able to be fixed with code development alone. If we believe the application cannot meet performance expectations in its current state, then upgrades to the frontend, backend, and/or database may be required in order to meet the test's passing conditions.

Functional tests will be run on completed features beginning in Sprint 4 and further functional tests will be used after Sprint 4 as more features are completed. Some features we expect to test in Sprint 4 are password reset functionalities, sorting cards by different criteria, and bookmarking cards. Performance tests will be run on a continuous basis as improvements are made throughout the remaining development process beginning in Sprint 4.

The deliverables for the testing process will be the final performance report generated by Lighthouse while running the application, the code for the unit test cases for each feature, and documentation detailing each test performed, the expected result of the test, and whether the test passed or not.

I.3. Scope

This document contains our strategy and plans for how our development team will test the newly added features and performance enhancements of the Living Atlas. The testing framework has yet to be implemented, so not all specific details for each test case are known at this time. This document's purpose is to outline a plan for testing the application for the development team as well as to inform the CEREO team of the development team's testing plans and allow for feedback for these plans.

II. Testing Strategy

The testing approach for the CEREO Living Atlas project focuses on core functionalities and new developments. It follows a structured methodology and will ensure all requirements listed in the 'Requirement and Specifications' document are met. During the testing process, Continuous Integration (CI) and Continuous Delivery (CD) practices will be used to continually refine test results and efficiently stage changes for deployment.

Multiple testing strategies will be employed, including unit testing, integration testing, system testing, and user acceptance testing. Each testing type ensures that individual components, their interactions, and the overall system function correctly. Frontend testing, in particular, can be automated using tools like Selenium or Jest. However, certain tests, such as logging into the Atlas, may require manual execution.

The detailed flow of our testing strategy can be found in Appendix A.

III. Test Plans

III.1. Unit Testing

Our team will follow all standard unit testing procedures. Most unit testing will focus on code developed by previous teams, as new development is mostly built using these previously developed units. The previously developed units to be tested include loading a card from the database, adding or deleting a card from the database, adding or deleting an account, submitting a form, and loading the map from Mapbox. It is crucial that these previously developed units are tested as the database host has been switched to Microsoft Azure and these units must still function with the new database for the overall system to function correctly. Some newly developed units to be tested include changing an account's password, sending a reset password email, and loading files and images from the database.

III.2. Integration Testing

Given the multi-layer design of our application, composing of a FastAPI backend, React Front end and integration with google cloud storage- the integration testing focuses on validating the flow of data between these layers. Unlike a monolithic structure, this modular setup introduces complexities in testing complete workflows and can cause issues with the flow of data between the layers.

The development team will primarily rely on manual testing within local and staging environments to replicate integrated scenarios like form submissions involving file and thumbnail uploads, database entry creation, and frontend content display. At this stage, automated integration testing remains limited due to the complexities of mocking Google Cloud APIs and maintaining database state across different layers. However, specific endpoints may still be tested using tools such as Postman or pytest with FastAPI's TestClient.

Although the ultimate goal is comprehensive end-to-end integration testing across all components, certain elements—like frontend thumbnail display logic—may be tested in isolation when backend deployment or effective mocking proves challenging.

III.3. System Testing

III.3.1. Functional testing:

Functional testing is carried out manually by developers, following the project's Requirements and Specifications document. Each functional requirement is matched with a specific test case. This includes:

- Creating a card with all metadata fields
- Uploading data files and thumbnail images
- Retrieving and displaying card data, including image previews
- Downloading uploaded files
- Deleting cards and confirming cascade deletions (files and thumbnails)

Tests are validated through the browser interface and network tools like Chrome DevTools or Postman. Any failed tests are documented with clear reproduction steps and assigned back to the original developer for resolution. As the application evolves, the testing plan will be updated accordingly.

III.3.2. Performance testing:

Performance testing targets two main areas: backend response time and frontend rendering speed.

Backend testing includes:

- Measuring FastAPI response times with large payloads (e.g., multi-megabyte uploads)
- Tracking latency when accessing files and thumbnails stored on Google Cloud

Frontend testing includes:

- Monitoring load times when rendering pages with numerous cards and images
- Evaluating performance on lower-end devices or slower networks

Manual observations are supported with browser profiling tools like Lighthouse and Chrome DevTools. While not benchmarked against strict numerical thresholds, performance is assessed qualitatively based on overall responsiveness and user experience.

III.3.3. User Acceptance Testing:

User acceptance testing will be conducted exclusively by the client. Instead of structured forms, feedback will be gathered through direct meetings.

A. Process

- The client will test key workflows, including creating, viewing, editing, and deleting cards.
- Particular focus will be placed on new features, such as image upload and thumbnail display.
- Observations and feedback will be shared in scheduled meetings.

B. Revision Process

- Developers will document and categorize feedback as bugs, feature requests, or general improvements.
- Actionable items will be tracked in the issue management system (e.g., GitLab Issues).
- Any necessary changes will be implemented and deployed to staging before final release.

The final iteration will be considered complete once the client confirms that all key features function as expected and the user experience meets their needs.

IV. Environment Requirements

For the frontend, we are planning to use React, which allows us to utilize tools like Jest and Selenium for testing. These tools facilitate easy testing of our UI across different browsers and JavaScript components. Lighthouse can be used to measure loading times and ensure performance standards are met. For staging environments, we are considering using Docker which will help create containerized versions of the application that can be easily deployed and tested across various machines. For the database, tools like Alembic or Flyway will manage schema changes and seed data during testing. We may switch to other tools as the project evolves, and the tools currently selected are subject to change. Version control is handled through GitHub. There are no specific hardware requirements. There are no specific hardware requirements for the testing environment.

V. Glossary

Lighthouse: An open-source Google-created developer tool that can run on a web page and generate a performance report with advice on how to improve performance

Jest: A testing framework used for JavaScript applications, often used to perform unit tests and ensure functionality works as expected

Continuous Integration: A software development practice where code changes are automatically tested and integrated into the repository as often as possible

Continuous Delivery: A software development practice where code changes are automatically tested and prepared for a release to production

Mapbox: A platform for designing and deploying custom maps and integrating these maps into web applications

Microsoft Azure: A cloud computing platform offering database hosting

FastAPI: A web framework for building APIs with Python

React: A JavaScript library for building user interfaces

Postman: An API development tool that can be used to build and test APIs
pytest: A Python testing framework that supports unit tests and integration tests, which can be used to test APIs

Chrome DevTools: A set of web developer tools integrated in Google Chrome that can be used to inspect and debug web pages and monitor performance

Google Cloud: A cloud storage service that allows for storage of data accessible over the internet

VI. References

Li, Eldon Y. "Software testing in a system development process: A life cycle perspective." *Journal of Systems Management* 41, no. 8 (1990): 23-31.

Appendix-A

Testing Strategy:

1. Review, revise, or update the Requirements and Specifications document to ensure alignment with the current project plan and processes.

2. Identify requirements to be tested based on the current Software Requirements Specification.
   1.
3. Determine which tests will be used to evaluate each module.

4. Review test data and test cases to ensure each unit is thoroughly verified and that the provided test data is sufficient to confirm proper functionality.
   1.

5. Define expected outcomes for each test.

6. Document the test configuration, test data, and expected results prior to conducting tests.

7. Conduct the tests, begin with unit tests. If a unit test is successful, the component will advance to integration/system testing.

8. If any test fails, log the issue clearly using a bug form. This form will detail the specific test case, the issue encountered, its possible cause, and the events leading up to the problem. Address the issue and repeat the testing process until it passes.

9. During testing, document all test data, test cases, and configurations used. Include this information in the revised Test Plan document.
   1.
10. After all tests and documentation are complete, submit the test documents and reports. Immediately handle any specifications that need further review, revision, or updating.