Ambitious Solutions

Software Testing Plan

Team Members:

Jasmine Flowers
Ian Nieto
John Gornick
Jerry “Tre” Kelley

Client: Jesslynn Stull
Mentor: Bailey Hall

Table of Contents

1. Introduction

The Altored Health web app is created to better educate the healthcare consumer population about the costs of healthcare before treatment. The main user population will be U.S healthcare consumers with and without health insurance. The main strength of the system is the search which will allow users to find their needed procedure and what that procedure should cost.

To test our software we will be performing unit testing, integration testing, and usability testing. For unit testing we will check the product piece by piece to make sure that the user from frontend to backend gets desired results. Next, integration testing will make sure that each individual piece of our software connects to each other piece flawlessly. Then, for usability testing we will more closely make sure current and future users find that the program fits their needs and functions the way they need it to. Finally we will, explain our methodology for deciding the criteria of what constitutes a functional product and how we will control the quality of the updates we decide to make to meet those requirements.

In this document you will find that our testing is focused on the data accuracy of the system as a priority because accuracy is the most important thing to our users. For our client if the accuracy is wrong the users may receive incorrect healthcare information which is a huge deal, and the company. We will be working to ensure that these issues don't become an issue by using the listed testing methods below.

2. Unit Testing within the Altored Health application

For our project unit testing will be focused on the core input and output given by the overall program and its components. This means that the user input will result in valid output, and each unit within our core project will contribute to the overall result with its set of correct information. Additionally in a slightly different form we will also expect the frontend to perform its functions correctly with things like correct formatting.

Units under test

1. **Backend Data Parser**
    - The Data parser will be responsible for getting and formatting the base set of information that the rest of our product relies upon. This information has to be correct for the rest of the product to work properly so this is the most important base test case.
2. **Backend to frontend data interpreter**

- This takes the input from the user and decides what data from the parser the user needs, and provides it to the **Data to user interface control subsystem.** This does use an ai subsystem in the current iteration so the information it comes up with will be considered correct if it seems correct to said user.

3. **Data to user interface control subsystem**
   - This system will take the user input and pass it to the **Backend to frontend data interpreter** and then take information from the **Backend to frontend data interpreter** and decide how it should be displayed to the user.
4. **Interface**
   - The interface is the all inclusive set of the main functions of our user interface besides the **Data to user interface control subsystem.** Since this is a startup and the format of this is ever changing we will focus on the main functions of the module as a whole.

Unit Test Design approach

1. **Backend Data Parser**

   Purpose: Convert healthcare information into a functional database table.

   Test Case Categories:

   - Correct parsed information: sets of insurance data turned into single sql datasets, and multiple cases.
   - Edge cases: Odd different sets of insurance providers, and lack of insurance providers.
   - Handles invalid datasets with missing important data

   Sample Tests:

   - Input dataset includes multiple procedures with differing insurances, and output condenses datasets while maintaining required information.
   - Input dataset includes insurance providers that are not formatted as expected, and the output either excludes edge cases, or knows enough to parse them too.
   - Input dataset has incorrect or incomplete information and the output does not include that information.
2. **Backend to frontend data interpreter**

   Purpose: This takes the input from the user and uses an ai to decide what data from the parser the user needs.

   Test Case Categories:

   - AI returns database information that is useful to the user based on the user's input.
   - Edge cases: User input's odd queries, and still gets valid information.
   - Detects invalid input and returns such.

Sample Tests:

- I need an Mri scan results in price and healthcare information about mri scans
- Bleaijdsaoijdaslkn results in things like mental health help or invalid input.
- %#$#@ results in invalid input.

3. **Data to user interface control subsystem**

   Purpose: This system will take the user input and pass it to the **Backend to frontend data interpreter** and then take information from the **Backend to frontend data interpreter** and decide how it should be displayed to the user.

   Test Case Categories:

   - Checks for valid characters to be passed to the backend and displays received information formatted correctly
   - Edge cases: odd numbers of returned healthcare insurances
   - Bad data: such as empty price categories

   Sample Tests:

   - I need an Mri scan results in formatted insurance information, pricing information formatted in user centric spaces.
   - A Rare procedure search results in relevant search results possibly missing insurance information if not covered by insurance and an explanation why that is.

4. **Interface**

   Purpose: The interface is the main user interaction space within the program, but not covering the **Data to user interface control subsystem.**

   Test Case Categories:

   - Users should be able to sign up, log in, and browse the webapp with ease.
   - Edge cases: Users may be unfamiliar with such systems
   - Bad connection errors

   Sample Tests:

   - A new user opens the app for the first time, it is easy to navigate and they are able to use the webapp.
   - A non tech literate user opens the app for the first time and has trouble but on site documentation explains use very quickly and easily
   - A user has some sort of connection error with the backend systems due to security issues, and the system suggests some fixes for their errors.

3. Integration Testing

3.1 Integration Testing in Our Project

Not every issue is part of a self-contained piece of software. After unit testing is done, there is still no certainty that the connections between different units are functional and allow work to be done correctly. Our application uses React on our frontend and Node.js as part of our backend, with a postgres database, a search API that connects to two AI services, and user authentication software. This poses many potential issues, like data mismatches, incorrectly formatted API responses, and potential contract violations between services. Our integration testing focuses on the gaps between these services. Primarily we want to ensure that we tackle all issues relating to search results, accuracy of results, or secure authentication.

3.2 Key Integration Points

Frontend ↔ Backend API

Our frontend is built in React, and communicates with our Node.js backend. This connection ensures that search queries are sent and structured results. Mistakes here could cause issues with data accuracy, data display, or interface issues.

Backend ↔ PostgreSQL

Our backend needs to connect to and properly parse and pull data from a secure database containing insurance pricing information. Issues here could result in inaccurate or missing pricing information, greatly impacting our user's experience.

Backend ↔ Search API

Our backend also needs to successfully connect to our search API, routing user queries and receiving structured results in return. Failures here could produce incorrect or incomplete search results being surfaced to the user.

Search API ↔ AI Services

The search API must correctly communicate with both AI services to process and interpret user queries. Failures here could result in inaccurate, irrelevant, or malformed responses that ultimately affect the quality of results displayed to the user.

3.3 Test Environment

Our tests run using github, representing a CI model of testing. These tests run automatically on every pull request. Significant choices of our testing include using the same database that we use during live demos. While this can sometimes be an issue, for the majority of applications our database is read only, allowing us to use the exact data we usually use with no risk. Test accounts

are configured locally in the user's environment. Cleaning our environment is never necessary, as no data is written during our tests, so there is little chance of corruption. We do rely on two external AI services, meaning our integration tests make live calls to these services rather than using mocked responses. This means test results for AI-related flows may vary depending on external service availability.

Task Scenarios

Integration Point: Frontend → Backend API → Search API → AI Services

3.4 Feature: Medical Procedure Search

Scenario Description: A user submits a natural language search query to find relevant medical procedures.

Integration Steps:

1. User enters a search query into the frontend search interface
2. Frontend sends a POST request to the backend with the query
3. Backend forwards the query to the search API
4. Search API sends the query to Groq, which translates natural language into medical codes
5. Codes are passed to Google Vertex, which structures them into app-compatible data
6. Results are returned through the backend to the frontend as a series of narrowing questions
7. Frontend displays narrowing questions to guide the user to a specific procedure

Expected Results:

- POST request is accepted and forwarded correctly
- Groq correctly translates the query into medical codes
- Google Vertex returns properly structured, app-compatible data
- Frontend displays relevant narrowing questions without errors

Failure Handling:

- Malformed POST requests return a clear error response
- AI service unavailability returns an appropriate error to the user
- Improperly structured data from Google Vertex is caught before reaching the frontend

Integration Point: Frontend → Backend API → PostgreSQL

3.5 Feature: Pricing Information Display

Scenario Description: A user selects a specific procedure and location to view insurance and self-pay pricing.

Integration Steps:

1. User completes the narrowing questions and selects a procedure and location
2. Frontend sends a POST request to the backend with the selected procedure and location
3. Backend queries the PostgreSQL database for matching pricing records
4. Database returns insurance and self-pay pricing data to the backend
5. Backend formats and returns the pricing data to the frontend
6. Frontend displays pricing information for relevant clinics and policies

Expected Results:

- Backend correctly queries the database with the right parameters
- Pricing data is accurately retrieved and formatted
- Frontend displays both insurance and self-pay pricing without errors
- Results are specific to the selected procedure and location

Failure Handling:

- Missing or malformed query parameters return a validation error
- No matching records returns an empty state rather than crashing
- Database connection failures return an appropriate error response

Integration Point: Frontend → Backend API → Microsoft Entra → PostgreSQL

3.6 Feature: User Authentication

Scenario Description: A user logs in through Microsoft Entra and gains authenticated access to the application.

Integration Steps:

1. User initiates login from the frontend
2. Frontend redirects the user to Microsoft Entra for authentication
3. Microsoft Entra validates credentials and returns an authentication token
4. Backend receives and validates the token
5. Backend establishes a user session and confirms access
6. Frontend grants the user access to the application

Expected Results:

- Microsoft Entra correctly validates user credentials
- Authentication token is returned and accepted by the backend
- User session is established successfully
- Authenticated user is granted full access to search and pricing features

Failure Handling:

- Invalid credentials are rejected by Microsoft Entra with a clear error
- Expired or invalid tokens are rejected by the backend
- Unauthenticated users are blocked from accessing search and pricing features

4. Usability Testing

Purpose

Functional correctness alone is not enough for a healthcare pricing platform. A system can return accurate data and still fail if users do not understand what they are seeing or cannot use the app effectively. Usability testing focuses on that gap. It ensures users can complete tasks correctly, efficiently, and without confusion.

For Altored Health, this is especially important. Users are often coming in with a real healthcare need and may already be stressed. They need to compare providers, understand different price types, and make decisions that affect their finances. If labels are unclear, flows are confusing, or something is hard to access, that is a real failure even if the backend is correct. Usability testing is how we catch those issues before release.

Method 1: Task-Based Think-Aloud Testing

Overview

In think-aloud testing, users are given realistic tasks and asked to explain what they are thinking as they complete them. The focus is not just on whether they succeed, but how they get there. This helps identify confusion, incorrect assumptions, and friction points in the interface.

Participant Selection

Participants should reflect the actual Altored user base. This includes people searching for medical procedures and comparing costs. It is important to include users with different levels of insurance knowledge.

Some users will understand deductibles and coverage. Others will not. Since the app presents both self-pay and insurance-based pricing, it needs to work for both

Around five users per round is enough. This is typically sufficient to uncover most major usability issues.

Task Scenarios

Tasks should match real use cases:

Find the lowest-cost provider within 50 miles for an MRI of your knee

With your saved insurance plan, find your estimated out-of-pocket cost for a colonoscopy at St. Luke’s

Search for City Radiology and find their self-pay price for a chest X-ray

Find a previous search in your history and open the results again

These cover the full workflow from search to interpretation to navigation.

Key Observations to Record

- Where users pause, hesitate, or backtrack
- Misinterpretation of pricing labels such as Standard Price vs Your Price
- Missing or unexpected UI elements
- Emotional reactions such as frustration or confusion

Success Metrics

- Task completion rate
- Time on task
- Error rate
- User satisfaction rating

Method 2: Heuristic Evaluation

Overview

Heuristic evaluation involves reviewing the interface against established usability principles such as Nielsen's heuristics. This does not require users and is useful for identifying structural issues early.

Heuristics Applied to Altored Health

- **Visibility of system status**
  The system should always show what is happening. This includes loading states, distance calculations, and quota status.
- **Match between system and the real world**
  Labels should be understandable. Terms like Gross Price or Personalized Rate can be confusing and should be evaluated carefully.
- **Error prevention**
  The system should prevent bad states. For example, if a user has hit their quota, the system should clearly communicate that instead of failing silently.
- **Recognition over recall**
  Users should not have to remember previous steps. Navigation should preserve context when moving between views.
- **Help and documentation**
  Tooltips should actually clarify meaning, not just repeat labels.

Evaluation Output

Each issue is assigned a severity score from 0 to 4. Issues rated 3 or 4 should be fixed before release.

Method 3: Accessibility Testing

Overview

Accessibility testing ensures the app works for users with disabilities, including those using screen readers or keyboard navigation. This is especially important for a healthcare platform.

Automated Accessibility Scanning

Tools such as axe can detect common issues:

- Missing ARIA labels
- Low color contrast
- Missing alt text
- Unlabeled form inputs
- Focus order problems

These checks should run in CI so issues are caught before code is merged.

Manual Keyboard Testing

Manual testing verifies:

- All interactive elements are reachable via keyboard
- Focus is always visible
- Modals trap and release focus correctly
- Navigation returns users to a logical position

Screen Reader Testing

Testing with tools such as NVDA or VoiceOver ensures:

- Prices are read with context, not just numbers
- Results and summaries are announced clearly
- Filters and sorting controls communicate state changes
- Distance and provider info are read in the correct order

WCAG Target

Altored Health targets WCAG 2.1 Level AA compliance, which is the standard baseline for accessibility in healthcare applications.

5. Testing Workflow and Quality Control

Overview

Testing in Altored Health is not a separate phase that happens after development. It is part of the development process. Every code change, from a small fix to a new feature, is expected to include tests that verify its behavior. The goal is to catch issues as early as possible, before they affect other parts of the system or reach users. This approach means writing tests alongside code, running them locally before pushing changes, and enforcing them through CI before anything is merged into the main branch.

Development Workflow

Work on Altored Health follows a branch-based development model. Each feature or bug fix is developed in its own branch, separate from main. This keeps unstable changes isolated and ensures that the main branch always reflects a working version of the system.

Before submitting a pull request, developers run the full test suite locally. This helps catch obvious issues early and reduces unnecessary CI failures. Once the branch is ready, it is submitted as a pull request to main. Each pull request requires at least one code review. This ensures that changes are not only correct, but also clear and maintainable. At the same time, automated tests run in CI. A pull request cannot be merged if any tests are failing. This is enforced strictly so that the main branch always remains stable.

Automated Testing in CI

All automated testing runs through GitHub Actions and is triggered on every pull request and push to main. The pipeline includes three main types of testing.

Unit tests verify individual functions, components, and API handlers. These include pricing transformation logic, search flow behavior, authentication handling, and UI rendering conditions. These tests are fast and provide immediate feedback on core logic.

Integration tests verify how different parts of the system work together. This includes communication between the frontend and backend, API interactions, and authentication flows. These tests ensure that the system behaves correctly as a whole.

Accessibility checks are included for user-facing components. Tools such as axe are used to detect issues like missing ARIA labels, low color contrast, and keyboard navigation problems. These checks run as part of the standard pipeline. If any test fails, the pull request is blocked from merging until the issue is resolved.

Quality Control Practices

Testing efforts are focused on the parts of the system with the highest impact on users. Pricing and data transformation logic is the most critical area. These functions must handle a wide range of inputs, including missing data, partial results, and edge cases. Errors here directly affect what users see, so they are tested extensively.

API and authentication logic is also a high priority. Tests verify request and response formats, authorization handling, and all failure conditions such as missing tokens or invalid roles. Since authentication affects all users, it must be reliable.

Error handling is treated as a core requirement. The system must respond clearly to issues such as API failures, invalid input, or quota limits. These scenarios are tested to ensure the system fails gracefully rather than breaking or returning misleading results. External services such as AI-based search APIs are handled through a service layer. These services are mocked in tests to keep results consistent. This ensures that test outcomes do not depend on external availability. Testing also uses realistic data structures based on actual API responses. This reduces the gap between test behavior and real-world usage.

Defect Tracking

All defects are tracked using GitHub Issues. Each issue includes a clear description of the problem, expected behavior, and steps to reproduce it. Issues are categorized by area such as pricing, authentication, or search. When a bug is fixed, a regression test is added to prevent it from happening again. This ensures that previously discovered issues do not reappear in future updates. Critical issues, especially those related to pricing accuracy or authentication, are treated as blocking. These must be resolved before other work continues.

Summary

This workflow ensures that testing is continuous and enforced at every stage of development. Automated tests validate each change, code reviews provide an additional layer of verification, and regression tests prevent repeated failures. The result is a system that remains stable as it evolves. For a healthcare pricing platform, this level of reliability is essential. Users depend on the system to provide accurate and understandable information, and this testing approach is designed to support that.

Conclusion

Altored Health is not just another web application. It deals with healthcare pricing, which means mistakes are not just technical issues, they can directly impact user decisions. If a user sees the wrong price, cannot access the system, or misunderstands what they are being shown, that is a real failure of the product.

The testing strategy in this document is designed to prevent those outcomes. Unit testing ensures that individual parts of the system behave correctly, including data processing, API behavior, and authentication. Integration testing verifies that those parts work together as expected across the full system. Usability testing ensures that users can actually understand and use the application, not just that it works technically. Accessibility testing extends that to users who rely on assistive technologies.

No single testing approach is enough on its own. A system can pass unit tests and still confuse users. It can look good from a usability standpoint and still return incorrect data. It can work well for most users and still fail completely for someone using a screen reader. Each layer of testing covers a different type of failure, and together they provide full coverage of the system.

All testing is integrated into a CI workflow so that every change is validated before it is merged. This reduces the risk of regressions and ensures that quality is maintained throughout development rather than checked at the end.

Overall, this testing plan is focused on two key goals: reliability and trust. The system needs to behave consistently, and users need to be confident in the information they are given. This plan ensures both.