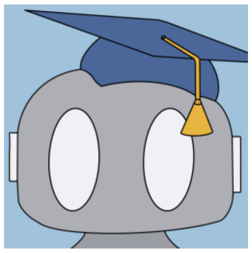
Personalized Education: An AI-Integrated Web Application for Personalized Tutoring

Software Testing Plan



Sponsors:
Dr. Andy Wang
Sethuprasad Gorantla

TutorTech:
Chase Babb
Ryley Fernandez
Faith Ononye
Shurie Kamewada

Mentor:
Paul Deasy

April 4, 2025

Table of Contents:

Introduction

TutorTech is developed for Metrology Research and Teaching Laboratory (MRTL) to personalize the learning experience of students as well as enhance student interaction with academic resources. This system allows students to securely sign up and log in, view course content, complete and submit assignments, receive grades, and track progress through an integrated dashboard. The primary goal of the platform is to optimize course engagement while incorporating AI-enhanced support and feedback.

Software testing plays an enormous role in ensuring this platform is reliable, secure, and intuitive for students and instructors. Testing validates both the system's correctness and the user experience, helping us identify issues in development. For this application, we will apply multiple forms of testing to ensure reliability and functionality. Unit testing will be used to verify that individual backend routes and logical units behave correctly. Integration testing will ensure that the major components of the system such as frontend UI, backend API endpoints, and database operations work together seamlessly. Usability testing will assess real user interaction and ensure the interface meets user expectations.

Our testing approach will begin with unit testing, focusing on individual backend routes and logic, such as user signup and login authentication. These units are some of the platform's core functionality, making them a priority for early validation. After verifying that each component performs as expected in isolation, we will proceed with integration testing across all major system boundaries, focusing on assignment submission, grading, chatbot integration, and session persistence which will evaluate how different modules interact with each other. Lastly, we will conduct usability testing, aimed at evaluating how effectively real users interact with the system. This phase will involve live demos and walkthroughs to gather necessary feedback.

We selected this structure to prioritize reliability in core functions (like authentication and submission) while also validating system-wide communication and user experience. The following sections will expand on our testing approach for each type of testing: unit, integration, and usability.

Unit Testing

Unit testing is the process of isolating and evaluating individual components or functions of a software system to ensure they behave as intended. These “units” are typically small, self-contained blocks of code such as a function, method, or route, that can be executed and tested independently from the rest of the system. The primary goal of unit testing is to validate that each unit behaves correctly across a variety of valid and invalid input scenarios, thus ensuring correctness, reliability, and stability.

For our application, unit testing plays a vital role in verifying the correctness of the backend API, especially for endpoints that directly impact the user experience and data integrity. Key operations such as user registration, login authentication, and assignment submission must work flawlessly, as these processes are core to user onboarding, content access, and grade tracking.

We use Pytest as our primary testing framework due to its ease of use, readability, and compatibility with Flask applications. Flask’s built-in test client allows us to simulate HTTP requests and responses in a controlled environment without needing to launch the entire frontend or browser stack. This enables us to verify response status codes, validate JSON response structures, and assert that proper database operations are performed.

To ensure robust testing, we designed test cases based on equivalence partitioning, boundary value analysis, and robustness testing. Each unit was tested across expected (valid) input ranges, boundary-edge cases, and erroneous or missing input to simulate real-world usage patterns.

1. User Signup (/signup)

Purpose:

Allow new users to register by submitting a unique user ID, valid email address, password, and personal details.

Equivalence Partitions:

- Valid: All required fields present, correct formats
- Invalid: Missing required fields (e.g., no password), duplicate user ID/email, malformed email

Boundary Values:

- User ID length: test maximum 8 characters (allowed) and 9+ (rejected)
- Password length: enforce reasonable minimum length (e.g., < 6 characters invalid)
- Email format: test edge cases such as “user@” or “user@com”

Test Scenarios:

1. Successful registration with all valid fields

2. Duplicate registration using existing user ID or email

3. Missing required fields (e.g., no password)

4. Invalid email format (e.g., “useremail.com”)

5. Password is hashed before storage

Tools Used:

- Pytest
- Flask test client
- SQL inspection for verifying hashed password format

Success Criteria:

- HTTP 201 returned on success
- HTTP 400 for duplicate or malformed inputs
- User data is stored correctly in student_information table
- Passwords are hashed securely using bcrypt
- Plaintext passwords are not stored or logged

2. User Login (/login)

Purpose:

Authenticate existing users using stored credentials and start a secure session.

Equivalence Partitions:

- Valid: Correct email/password

- Invalid: Incorrect password, unregistered email, missing fields

Boundary Values:

- Short passwords (below minimum)
- Invalid data types (e.g., numbers instead of strings)

Test Scenarios:

1. Login with correct credentials (email/password)

2. Incorrect password

3. Login with non-existent email

4. Missing fields (email or password)

5. SQL injection attempt (e.g., "admin' OR 1=1")

6. Login fails if bcrypt hash does not match input password

Tools Used:

- Pytest
- Flask test client
- bcrypt.checkpw() for password verification

Success Criteria:

- HTTP 200 returned with user info and session cookie on success
- HTTP 401 returned for all failed login attempts

- No user data is exposed on failure
- Login only succeeds if bcrypt.checkpw() validates password

3. Assignment Submission (/api/assignments/<id>/submit)

Purpose:

Allow students to submit assignment answers and receive feedback with a calculated score.

Equivalence Partitions:

- Valid: All answers provided, valid assignment ID
- Invalid: Missing answers, wrong assignment ID, unauthenticated request

Boundary Values:

- Empty or partial answer object
- Very long text answers
- Incorrect data types (e.g., numbers instead of strings for text answers)

Test Scenarios:

1. Submit complete assignment with all answers

2. Submit with no answers or some skipped

3. Submit to non-existent assignment ID

4. Submit without user ID or session

Tools Used:

- Pytest
- Mock submission payloads
- DB inspection for grades table entries
- Assertion of grading feedback in response

Success Criteria:

- Assignment is auto-graded and score is returned
- Grade is saved in the database
- HTTP 200 or 201 response with result details

These units represent the foundational flow of a user in the system. Without reliable user registration or login, no access to courses or content is possible. Assignment submission is a core educational feature, tied directly to feedback and learning progress. Any issues in these routes would cause significant usability and data reliability concerns.

We chose not to include unit tests for backend features tied to Qdrant memory storage or AI response generation, as these rely on third-party APIs, embedding models, and semantic vector searches. These are better covered in integration testing, where their interaction with other components can be validated in full-system context.

Our unit testing approach combines rigorous scenario design (using equivalence partitions and boundary value analysis) with simple yet effective tools like Pytest and Flask's test client. Each test simulates a realistic request to an isolated API route and checks for both expected and erroneous behaviors. These tests ensure system stability early in the pipeline, providing a dependable foundation for broader integration and usability testing later on.

Integration Testing

Integration testing is a critical part of the software testing life cycle. This particular aspect is focused on ensuring that all individual modules of an application function correctly when working in tandem. To be more specific, integration testing evaluates the interactions between different components; and is especially concerned with data flow, communication, interface interactions, etc. For our TutorTech platform, integration testing is especially important to ensure all of the intricate pieces function accordingly. The system's complexity arises not just from the internal logic, but also from how the modules exchange data across the client-server boundary, interact with persistent databases, and manage user context throughout protected routes. Special focus was placed on preserving user experience through persistent state and enabling AI personalization. By validating these cross-cutting concerns, we established confidence in the "plumbing" that powers our LMS-like tutoring system. A few main goals of integration testing in this application were:

- To verify that the data passed between the frontend and backend is accurate and correctly structured.
- To ensure that user authentication and session persistence are consistently enforced across protected components.
- To validate that dynamic content loading (e.g., enrolled courses, assignment data, chatbot responses) is executed properly with backend data sources.
- To confirm that the chatbot and learning preferences system integrates smoothly with stored user profiles and delivers customized responses.

The Integration testing approach was both manual and automated (where applicable) and prioritized boundaries between the following key module pairs:

1. React components & Flask API endpoints
2. Frontend forms & backend validation/storage (e.g., preferences, login)
3. Protected routes & session state (via UserContext)
4. AI Chat system & Qdrant database memory
5. Assignment submission & AI-enhanced grading logic
6. Data retrieval methods & Relational database.

Each test followed a specific procedure. First, the interface was identified (data input / output). Next, the assumptions and expectations were defined and formatted for that specific interaction. After that, either mock or real user input was used to trigger a subsequent reaction. From there, the system response was observed (success, failure, or error), and finally, the results are logged and / or the mismatch between modules is resolved. A more in-depth integration test plan for the individual modules can be observed below:

1. User authentication

   Integration point: Login.js, UserContext.js, ProtectedRoute.js, and /login endpoint.

   Objective: Ensure users can log in and access protected routes using context from the backend to preserve the session.

   Test Steps:

   - Submit valid credentials from the login form.
   - Verify that a session cookie is stored and UserContext is updated.
   - Attempt to navigate to /dashboard and ensure access is granted.
   - Log out and confirm redirection to /login.

   Test Harness: Browser DevTools and mocks for session API

   Success Criteria:

- useUser() hook returns populated user object after login.
- ProtectedRoute component renders children only when user context is populated.

2. Learning preference and submission

   Integration point: LearningStyleQuiz.js to /save-preferences to PostgreSQL column learning_preferences.

   Objective: Ensure user selections in the learning style quiz are saved in the database and used later for AI customization.

   Test Steps:

   - Select options in the quiz and submit.
   - Intercept the network request to verify correct payload.
   - Check the database for saved preferences using SQL or print logs.
   - Navigate to the chat page, select "Custom" bot mode, and confirm the generated prompt reflects preferences.

   Test Harness: Browser DevTools, backend logging, test API endpoints.

   Success Criteria:

   - JSON-encoded preferences are stored accurately.
   - AI prompt dynamically changes based on saved preferences in generate_prompt_from_preferences().

3. Assignments and automatic grading

   Integration Point: Assignment.js to /api/assignments/<id>/questions and /submit.

   Objective: Verify full interaction loop: assignment loads, the answers are submitted to the AI grading, the results are returned and are then stored in the database.

   Test Steps:

- Load a course page and begin an assignment.
- Submit various question types (multiple choice, text).
- Confirm backend processes grading and returns per-question feedback.
- Check /api/grades/<user_id> endpoint for updated score.

Test Harness: Console logs for AI grading, mock responses, database inspection, and UI assertions.

Success Criteria:

- Each answer maps to a question ID and is properly graded.
- AI feedback reflects question keywords.
- Final score updates on frontend and backend persist.

4. Courses / Dashboard

   Integration Point: Courses.js, Dashboard.js, and /enrolled-courses/<user_id>

   Objective: Validate the user's enrollment status and that their dashboard reflects current course enrollments.

   Test Steps:

   - Enroll a user in a course via /enroll.
   - Visit /dashboard and confirm that only enrolled courses are listed.
   - Confirm /courses show correct status ("Enrolled" or "Enroll" button).

   Test Harness: Manual UI validation and console logs for fetched data.

   Success Criteria:

   - Dashboard shows only enrolled courses tied to the current user session.
   - State updates immediately after enrollment.

5. AI chat and memory search (both recent and semantic)

Integration Point: Chat.js to /api/chat to Qdrant memory layer.

Objective: Ensure messages from the frontend are sent to the backend, semantically embedded, and stored in Qdrant for future context use.

Test Steps:

- Send a question via the chat component.
- Confirm the backend receives and embeds message.
- Log memory vector insertion.
- Submit a vague follow-up and confirm relevant message pair is injected using vector similarity.

Test Harness: Backend logs, manual chat UI testing, Quadrant CLI/API calls (or just vist Qdrant dashboard)

Success Criteria:

- Chat response includes contextually related memory entries.
- Message embeddings stored and retrievable from Qdrant.

6. Admin communication via contact us

Integration point: Contact.js to EmailJS API

Objective: Ensure that messages submitted via the Contact form are correctly sent through the EmailJS service and the user receives visual confirmation of success or failure.

Test Steps:

- Navigate to /contact and trigger the modal by clicking the envelope icon.
- Fill out the form fields: name, email, subject, and message.
- Submit the form and intercept the request sent to EmailJS.

- Verify success state (modal closes, user is redirected, and success alert is shown).
- Simulate an error by entering invalid EmailJS credentials or disconnecting the network, then verify that an appropriate error message is displayed.

Test Harness: Manual UI interaction, console monitoring and browser DevTools (for request/response tracking).

Success Criteria:

- All form fields are validated and required before submission.
- The form uses the emailjs.sendForm() method with correct service and template IDs.
- On success, the user is shown an alert and redirected to the homepage.
- On failure, an error message is logged and displayed to the user.
- No sensitive data is stored; messages are only passed through to the external mail service.

Usability Testing

Usability testing focuses on evaluating how real users interact with the system, and whether the interface, features, and workflows are understandable, efficient, and accessible. It is particularly important for end-user-facing platforms like TutorTech, where students and instructors may interact with complex functionality such as course enrollment, assignment submission, and AI chatbot tools. The goal is to identify friction points, improve the user experience, and ensure that users can achieve their goals with minimal confusion or frustration.

For TutorTech, usability is central to success. Our target users include students and instructors within the Metrology Research and Teaching Lab (MRTL), who are generally tech-savvy due to the advanced nature of their coursework. However, the platform still needs to be intuitive for first-time users, especially when navigating between dashboards, accessing assignments, and interacting with the AI bot. Our testing is designed to reveal interface ambiguities, layout inefficiencies, and areas where users may require more guidance.

Feedback Sources and Justification

Our usability testing strategy includes feedback gathered from regular meetings with our project client and mentor, as well as in-class presentation sessions where we showcased prototypes and core features. These meetings function similarly to expert reviews, offering insights from stakeholders with technical and instructional experience. Our client has provided direction on what the platform should prioritize in terms of clarity and accessibility.

We also proposed including a QR code on our final presentation, inviting students to sign up and explore the platform. This would allow us to observe how new users register, navigate, and use the system organically. Although this was not executed during development due to time constraints, it remains part of our long-term usability testing vision. We anticipate recording their interactions (with consent) and gathering informal feedback about clarity, speed, and perceived usefulness.

Core Testing Areas

1. **Signup/Login Experience**
    - Goal: Ensure that new users can successfully register and log in without encountering unclear input fields or validation errors.

- ○ Evaluation: Observing whether users understand form labels, receive clear error messages, and are redirected correctly.

2. **Navigation Between Pages**
   - ○ Goal: Confirm that users can move fluidly between dashboard, courses, assignments, and chatbot.
   - ○ Evaluation: Measure time taken to locate desired features; track user confusion or hesitation.
3. **Assignment Workflow**
   - ○ Goal: Evaluate the clarity of assignment questions, answer input fields, and the grading feedback.
   - ○ Evaluation: Observe whether users complete submissions without guidance and understand their results.
4. **Chatbot Interaction and Response Time**
   - ○ Goal: Assess how effectively the AI chatbot supports learning and whether its response time affects user satisfaction.
   - ○ Evaluation: During presentations and demos, users interacted with the chatbot and we recorded how quickly it responded. If delays occurred, we noted user reactions. Timely and relevant feedback is essential to ensure AI integration does not hinder usability.
5. **Error Handling and Alerts**
   - ○ Goal: Confirm that all error states (e.g., missing answers, invalid input, failed submission) are gracefully handled.

  - Evaluation: Observe whether users are confused when errors occur or if the UI communicates the issue effectively.

6. **Accessibility and Visual Design**
   - Goal: Review the clarity of fonts, spacing, colors, and overall layout.
   - Evaluation: Gather feedback on ease of reading, contrast, and responsiveness on different devices.
7. **Scalability and Load Conditions**
   - Goal: Evaluate whether the platform remains usable when multiple users are logged in simultaneously, such as during classroom-wide access or demo presentations.
   - Evaluation: Monitor page load times, chatbot responsiveness, and database-driven queries to ensure no performance bottlenecks.

Usability Testing Strategy and Timeline

- **Weekly Check-ins with Client and Mentor:**
  - Duration: Entire development cycle
  - Method: Live walkthroughs, visual feedback, discussions about unclear or missing features
- **In-Class Presentation Testing:**
  - Duration: Mid-development and final phase
  - Method: Simulated user interactions and live demos
  - Output: Notes on interface issues, success stories, and feature requests
- **QR Code-Based Open Testing (Future Plan):**
  - Duration: Final project showcase

- Method: Invite students to test the platform organically via a QR code on poster
- Output: Informal feedback on usability, signup clarity, and user flow

All feedback will be compiled and reflected upon in our final evaluation, and major themes will be used to justify post-launch improvements.

By covering these usability dimensions, from first impression and task clarity to system performance under realistic load, we ensure TutorTech delivers not only functional correctness but also a supportive, smooth learning experience tailored to our user base.

Conclusion

TutorTech is designed with the goal of enhancing student engagement through AI-supported learning, personalized feedback, and seamless interaction with course content. As such, our testing strategy has been carefully structured to reflect these priorities, ensuring that critical functionality is reliable, integrations across components are seamless, and user experiences are smooth and intuitive.

We began by validating individual backend routes through **unit testing**, focusing on essential processes such as user signup, login authentication, and assignment submission. These endpoints are foundational to the system's functionality and user trust, and our tests accounted for equivalence classes, boundary conditions, and robustness against invalid inputs.

We then expanded our scope through **integration testing**, where we evaluated the flow of data and logic across components such as the frontend UI, backend Flask APIs, and PostgreSQL database. Particular emphasis was placed on session persistence, AI-based chatbot communication, automatic grading, and EmailJS integration. These tests gave us confidence that the different parts of our platform interact consistently and correctly.

Finally, we laid out a thoughtful **usability testing** plan that incorporates real feedback from clients, mentors, and end users. We focused on clarity, workflow, and responsiveness, especially during in-class demos and live walkthroughs, and proposed further testing opportunities at our project showcase through QR code access. Additionally, we considered the platform's scalability, planning to observe user interaction under load and assess response times for time-sensitive features like chatbot feedback.

Our testing process ensures that TutorTech is robust, user-centered, scalable, and ready to support students and faculty in a real-world academic setting. Through this multi-layered strategy, we’ve aligned our testing practices with the educational goals and technical needs of MRTL.