Software Testing Plan

Version 1

March 20th, 2026

CLASS (Cybersecurity Learning with AI for Static Systems)

Team Members: Sean Golez, William Barnett, Kayden Vicenti, Colton Leighton

Team Mentor: Scott LaRocca

Sponsor: Dr. Lan Zhang

School of Informatics, Computing, and Cyber Systems, Northern Arizona University

Table of Contents

Introduction

CLASS is a web-based AI tutoring platform designed to support students who are learning about cybersecurity through the use of static analysis capture the flag challenges (CTF). Static analysis CTFs are widely used to teach secure software design, vulnerability discovery, and program reasoning. However, these challenges are difficult for learners because they require an understanding of control flow, data flow, and program semantics without executing the program. In instructional settings, students depend on instructor and TA assistance, and the use of general-purpose AI tools. These tools provide inconsistent and inaccurate guidance due to the lack of grounded, authoritative content. Sponsored by Dr. Lan Zhang of Northern Arizona University and developed in support of the SE450 course, CLASS addresses this gap by providing scalable, structured, and conceptually grounded tutoring for the static-analysis CTF labs.

The current cycle of this project involves testing, ensuring the system behaves correctly regarding the architecture, frontend, backend services, knowledge graph, and AI integration. The system consists of several components: a React-based frontend, a Python backend, a Neo4j knowledge graph database, and an external LLM service. Testing will occur in both local development environments and the containerized environments using Docker. The scope of testing will involve user authentication, document upload, interaction with the chatbot, and knowledge graph retrieval. Other areas will include non-crucial UI testing and the accuracy of the responses from the LLM. Some of our planned testing strategies include: unit, integration, usability, and security testing. Previously and currently, testing and reviews have been conducted throughout the development of our project. Through the review of each pull request and testing before alpha demos.

The strategy our team has for testing allows our system to address the complexity of our project while reducing risks. The backend logic, authentication, and AI responses require additional testing due to the need for correctness related to our project. UI elements will be tested lightly and, through the use of usability testing, will be validated. The following sections will describe each testing type in detail.

Unit Testing

For our project, we define unit testing as a way to verify the correctness of each component within our application, such as functions and modules. In CLASS, unit testing will focus on the backend logic, data processing, and API responses. For unit testing, our project will have the following tools and metrics for testing. Python’s pytest framework, JavaScript for the front end using Jest, and for this, we will be testing locally during development. Some metrics will include test pass/fail, code coverage, and focusing on modules that can be considered high risk, such as authentication and data processing.

LLM Testing:

To validate the correctness of the LLM response, we need to define metrics that accurately evaluate the results. Because we cannot directly unit test the LLM, we have to reference the queries with an expected answer criterion. The responses will be evaluated against the following rubric. We will create 15-20 questions related to the course documents and their correct responses, and for each question, we will note the concepts used and if any hallucinations occur. However, there are some limitations, such as identical questions may yield different results or wording. Below will outline some main goals and examples of testing scenarios.

Testing Goals:

Correctness: Ensure responses are factually accurate and aligned with cybersecurity concepts. Prevent misleading, hallucinated responses
Relevance: Ensure responses address the user's question appropriately
Clarity: Ensure explanations are understandable for the target audience
Consistency: Ensure similar prompts produce reasonably consistent outputs

1. Scenario-Based Testing: These tests will simulate student interactions with the chatbot.
   Scenario:
   a. User asks a cybersecurity question
   b. System retrieves relevant knowledge graph data
   c. LLM generates a response using context
   d. Response is evaluated for correctness and clarity
   Expected Outcomes:
   a. Response uses relevant context
   b. No hallucinated or incorrect facts
   c. Explanation is appropriate for the student's level
2. Prompt-Based Testing: A predefined set of prompts will be created based on course materials and expected student questions.
   Prompt Categories:
   a. Conceptual questions: What is static analysis?
   b. Procedural questions: How do I use this tool?
   c. Edge cases: ambiguous or incomplete questions
3. Human Evaluation (Manual Review): Reviewers will assess LLM responses using a scoring rubric.
   Evaluation Criteria:
   a. Accuracy: Correct, Partially Correct, Incorrect
   b. Relevance: Fully, Partially, Not relevant
   c. Clarity: Clear, Somewhat unclear, Confusing
   d. Helpfulness: Helpful, Neutral, Unhelpful

Units Under Testing:

The following modules and functions will be used for unit testing with an emphasis on risky areas.

Module	Functions under testing	Rational
db.py	Hashpassword, verifypassword	Allows testing for security logic and errors that may compromise user data
rag.py	Rag pipeline initialization, prompt template assembly, response generation	Controls information related to the education of the LLM output
kg.py	getFullGraph, node/edge query construction	Data integrity that propagates to LLM

UI rendering components such as Next.js and React, OpenAI model, and Neo4j will be excluded from unit testing. These are because the system is outside control, or presentation-only code with no business logic.

Unit Test Design

1. Unit Under Test: HashPassword & VerifyPassword
    a. Purpose: Produces a hashed password and verifies its correctness
    b. Test Case Categories:
        i. Valid inputs: Alphanumeric, with special characters
        ii. Boundary cases: Minimum length (12 characters), maximum length password (70 characters)
        iii. Invalid inputs: Empty string, null value, non-string types
    c. Sample Tests:
        i. HashPassword(“ThisisApassword123!”) will return a non-empty string not equal to the original string
        ii. HashPassword(“”) raises a ValueError or returns a handled error
        iii. VerifyPassword(“password1”, hash) returns true for the correct password
        iv. VerifyPassword(“WrongPass”, hash) returns false

2. Unit Under Test: Signin
    a. Purpose: Authenticates the user and returns their role
    b. Test Case Categories:

        i. Valid inputs: Username and password for current student/instructor accounts
        ii. Boundary cases: Username of maximum length, and max length password
        iii. Invalid inputs: Nonexistent username, existing correct username with wrong password, missing fields
    c. Sample Tests:
        i. Valid student credentials return HTTP 200, and role=student
        ii. Valid instructor credentials return HTTP 200, and role=instructor
        iii. Nonexistent username returns HTTP 401
        iv. Correct username with incorrect password returns HTTP 401
        v. Missing password field returns HTTP 422
3. Unit Under Test: Getfullgraph
    a. Purpose: Retrieves nodes and relationships from the Neo4j knowledge graph
    b. Test Case Categories:
        i. Valid inputs: Graph with multiple nodes and edges
        ii. Boundary cases: Empty graph(no nodes), graph with nodes but no relationships
        iii. Invalid inputs: Database connection failure, or warped query result
    c. Sample Tests:
        i. Returns a correctly structured graph with nodes and edges, and a populated graph
        ii. Returns empty lists for nodes and edges when the graph is empty
        iii. Raises handled exception
4. Unit Under Test: Prompt Assembly
    a. Purpose: Assembles the final prompt sent to the LLM from the retrieved graph context, the user message, and the instructor prompt
    b. Test Case Categories:
        i. Valid inputs: Valid retrieved context strings, valid user message, non-empty instructor prompt
        ii. Boundary cases: Empty retrieved context, empty instructor prompt
        iii. Invalid inputs: No values for context or user message, long inputs exceeding token limits
    c. Sample Tests:
        i. Assembled prompt contains the user message, context, and instructions
        ii. Empty instructor prompt, prompt still contains context and user message without errors
        iii. With no retrieved context, the prompt defaults to the fallback instructor and does not crash

Integration Testing

Integration testing in CLASS verifies that the individual components interact correctly when combined. Following the Alpha 1 demo, the system now has active connections between Next.js frontend, FastAPI backend, Neo4j knowledge graph, and session store using PostgreSQL and the OpenAI LLM. Integration testing will target the connection or boundaries between these components, where data formats mismatch, authentication errors, or failures may surface.

Key integration points:

1. Correct authentication and role assignment
2. Chat generation should produce correct nonempty responses
3. Knowledge graph retrieval queries should produce the correct information context for LLM
4. Document upload should update without failures
5. Multichat sessions require accurate storage and retrieval
6. Custom prompts must persist and be correctly injected into LLM calls

Integration Scenarios:

Feature: User authentication and role-based UI

Scenario Description: login and role conditional navigation

Integration Steps:

a. User submits the sign-in form with valid student credentials
b. Backend queries PostgreSQL, verifies password hash
c. Frontend stores role in localStorage and renders UI
d. Repeat with instructor credentials

Expected Results:

e. Frontend sends POST to /signin
f. Returns HTTP 200 with role="student"
g. Student view displayed; Knowledge Graph button hidden
h. Instructor view displayed; Knowledge Graph button visible

Failure Handling:

i. Invalid credentials return 401, and an error message is shown. Missing fields return 422. No role-specific content is accessible without a successful login.

Feature: End-to-end RAG response

Scenario Description: Student submits a question and receives a tutoring response

Integration Steps:

a. User enters text to the chat through the frontend
b. Frontend sends a POST request to the backend API with the query
c. Backend validates the input, retrieves context from the knowledge graph, and streams the OpenAI API response to the frontend
d. Frontend displays the response as it is being generated

Expected Results:

a. Backend accepts the POST request from the frontend
b. OpenAI API correctly generates the response
c. Backend correctly streams the response to the frontend
d. Frontend UI correctly displays the response as it is being streamed

Failure Handling:

a. Invalid OpenAI API key
b. Invalid Neo4j authentication
c. Invalid chat id returns 404

Feature: Instructor knowledge graph curation

Scenario Description: Teacher edits the knowledge graph using the curation interface

Integration Steps:

a. User adds, deletes, and/or edits node and relationship informations
b. User saves changes from the frontend UI
c. Frontend sends a POST request to the backend API with node and relationship information
d. Backend merges the data with a test Neo4j database
e. Backend returns a successful response
f. Frontend displays that information was saved successfully

Expected Results:

a. Backend accepts the POST request from the frontend
b. Backend successfully merges data with the Neo4j database
c. UI reflects the successful creation without errors

Failure Handling:

a. Invalid Neo4j authentication
b. Error while saving returns 500

Feature: Document upload and generation

Scenario Description: Teacher uploads a document, and a knowledge graph is generated

Integration Steps:

a. The teacher uploads a document from the frontend UI on the knowledge graph view
b. Frontend sends a POST request to the backend API with documents
g. Backend undergoes exception handling to add the uploaded file to the file location
h. Backend processes the document file paths through a knowledge graph builder pipeline and create the necessary indexes for document search
i. The backend process will remove the temporary files from storage as a cleanup and return a success prompt
j. Frontend receives response and updates the UI, displaying the newly generated knowledge graph

Expected Results:

a. Backend accepts the POST request from the frontend
b. Backend successfully adds a temporary file path for the documents
c. LLM accurately manages knowledge graph curation
d. Backend successfully removed temporary file paths
e. Frontend updates appropriately to reflect the changes within the database from document upload

Failure Handling:

a. Invalid documents that were uploaded result in a caught error handling and are returned to the user

Feature: Multi-session chat persistence

Scenario Description: A user creates, revisits, and continues multiple chat sessions, with conversation history stored and retrieved across sessions.

Integration Steps:

a. User opens the chatbot interface and starts a new chat session.
b. Frontend generates a unique chat_id and associates it with the session.
c. User submits a message; frontend sends the message and chat_id to the backend API.
d. Backend logs the message in a test database using log_message()
e. Backend retrieves prior messages for the chat_id (if any) via retrieve_history()
f. Backend sends conversation context to the LLM for response generation.

g. LLM generates a response, which is streamed back to the frontend and stored in the database.
h. User revisits a previous session; frontend requests the stored chat history for that chat_id.

Expected Results:

a. Frontend successfully creates and tracks a unique chat_id per session.
b. Backend stores all user and assistant messages under the correct chat_id.
c. Chat history is accurately retrieved when revisiting previous sessions.
d. LLM receives the correct conversation context for each request.
e. Frontend displays complete, consistent chat history without mixing sessions.

Failure Handling:

a. Missing or invalid chat_id triggers creation of a new session or an error message.
b. Database write or retrieval failure return error messages and prevent inconsistent session state.
c. LLM or streaming errors are caught and displayed to the user without breaking ongoing sessions.

After verifying that the system components interact correctly from a technical perspective, it is also important to evaluate how users will experience and interact with the system as a whole. The following section focuses on the system functionality for a user through usability testing.

Usability Testing

Usability testing focuses on whether students and instructors can effectively and comfortably use the system (the chatbot and the knowledge graph) to support their learning during static-analysis CTF challenges. For our tool, it will be important to make sure users can navigate the interface, ask questions naturally, receive correctly interpreted responses, and access helpful supporting information, while also enabling instructors to curate and guide responses through the knowledge graph and custom prompts.

The main goals of this testing are to verify that a student, typically with some programming or cybersecurity learning goals in mind and limited experience with static analysis or AI tutoring, can use the chatbot to help complete a static-analysis CTF challenge. Another goal is that a teacher can use both the chatbot and the knowledge graph to curate chatbot responses using selected documents or custom prompts. The target teacher is an instructor familiar with the course material but new to AI tutoring, seeking to provide students with an innovative way to learn about static-analysis CTF challenges. These factors led us to choose a

practical usability testing approach with real students and representative instructors, performing realistic tasks under observation.

The overall system is not expected to be highly complex for students, as their interaction is primarily limited to the chatbot interface, while instructors interact with both the chatbot and the knowledge graph, making their workflow slightly more involved and potentially more time-consuming. To better understand these differences, we will record the time it takes for both students and instructors to complete their respective workflows, not to enforce strict performance benchmarks, but to provide context for identifying friction points. Potential consequences of poor usability include students being unable to effectively complete CTF challenges using the chatbot, or instructors being unable to successfully curate responses through document selection or custom prompts. In either case, such issues would indicate areas that require refinement in subsequent development cycles.

We will use short task-based usability sessions, informal client and mentor feedback, and internal reviews for testing. As for the student workflow, we will have at least one cybersecurity student use C-LASS while completing a static-analysis CTF challenge to see whether the chatbot is clear, useful, and easy to use in a realistic setting. The instructor workflow testing will involve evaluating whether a simulated instructor can use the chatbot and knowledge graph to guide responses by inserting documents and custom prompts into the knowledge graph view, and whether or not the results of the curation are accurate to the simulated teacher's use case. Usability testing tasks for both student and teacher modes will include logging in, asking questions, following up for clarification, and adjusting response behavior. Feedback will be collected through observation, user comments, and success/failure metrics of task completion.

Usability testing will take place throughout development rather than only at the end. After Alpha II, we will test basic core workflows such as login, navigation, and chatbot interaction. Before Alpha III, we will test more specific student and instructor workflows, including CTF tutoring use for students, and response curation through the knowledge graph and custom prompts for instructors. Findings will be documented, prioritized by impact, and addressed in subsequent development cycles to continuously refine usability and user experience.

Student Workflow Usability Testing

Users: Cybersecurity students (primary), mentor/client reviewers (secondary)

Goals: Ensure the chatbot supports CTF problem-solving in a clear, intuitive, and effective way, with minimal confusion or misuse.

Method: Short, task-based usability sessions using a real static-analysis CTF challenge, along with observation and informal feedback.

Sessions:

- 1–3 cybersecurity or software engineering students
- 1 mentor/client feedback session

Tasks:

- Log in and navigate to the chatbot interface
- Ask questions related to a static-analysis CTF challenge
- Interpret and apply chatbot responses to progress in the challenge
- Ask follow-up questions for clarification
- Adjust prompts to refine responses

Measures: Task completion rate, number of errors/confusion points, effectiveness of responses, user comments.

Follow-up: Prioritize high-impact usability issues (e.g., unclear responses, navigation friction) and address them in the next development cycle.

Instructor (Simulated Teacher) Workflow Usability Testing

Users: Simulated instructors (primary), mentor/client reviewers (secondary)

Goals: Ensure instructors can effectively curate and control chatbot responses using the knowledge graph and custom prompts.

Method: Task-based usability sessions simulating instructor workflows, combined with internal reviews and feedback.

Sessions:

- 1–2 simulated instructor sessions (team member acting as instructor)
- 1 mentor/client feedback session
- 1 internal expert review

Tasks:

- Log in and access the knowledge graph view
- Insert and manage documents within the knowledge graph
- Add and modify custom instructor prompts
- Use the chatbot to verify that responses reflect curated content
- Adjust inputs to improve response accuracy and alignment

Measures: Task completion rate, accuracy of curated responses, number of errors/confusion points, user comments.

Follow-up: Identify issues in response control, prompt usability, and knowledge graph interaction, and prioritize fixes for upcoming development iterations.

Testing Workflow and Quality Controls

Our group uses GitHub to track our progress and code review to identify any issues with the implementation. Using GitHub issues will be beneficial for our project as we can describe an issue through the use of labels like severity, expected results, and possibly the type of environment that an issue was identified with.

Severity	Definition
Low	Minor issues
Medium	The feature partially works
High	Major feature broken, but has some functionality
Critical	System crash/data loss/complete failure that hinders progress

This structured outline allows testing processes and quality control within the project by identifying and addressing issues as they arise. By using this table, we are able to maintain our project progress by outlining the standards for a defect.

Conclusion

Our test plan outlines a comprehensive approach to smoothly integrating, verifying the correctness, and usability of our application. Unit tests will target the business logic and authentication layer and the RAG pipeline ensuring that the main functionality is individually validated before integration. Our integration testing will validate the data flow between the frontend, backend, knowledge graph, and chat session. The usability testing ensures that the system serves users effectively. This testing outline establishes an outline to classify issues and allows our team to address the problem based on categorization. Together, these main testing methods will allow us to properly implement the client's main functionality needs while minimizing risks associated with implementation.