Capstone Design and Verification & Validation Plan

Group 7 - LM Compass

COMPSCI 4ZP6

Dr. Mehdi Moradi - Dr. Angela Zavaleta Bernuy

January 23, 2026

Sohaib Ahmed	ahmes127@mcmaster.ca	Coordinator/Researcher
Madhav Kalia	kaliam1@mcmaster.ca	Backend Engineer
Aadi Sanghani	sanghana@mcamster.ca	Backend Engineer
Gulkaran Singh	singg36@mcmaster.ca	Fullstack Engineer
Rochan Muralitharan	muralr3@mcmaster.ca	Fullstack Engineer
Owen Jackson	jackso4@mcmaster.ca	AI Engineer
Aryan Suvarna	suvarnaa@mcmaster.ca	Frontend Engineer

Table of Contents

1. Revision History

Date	Version	Authors	Description
2026-01-23	1	Group 7 (Sohaib, Madhav, Aadi, Gulkaran, Rochan, Owen, Aryan)	Initial Design + V&V Document

2. Purpose of the Project

2.1 Background

Generative AI is expanding rapidly, but evaluating the quality of responses from Large and Small Language Models (LLMs/SLMs) remains a challenge, so it's difficult to understand which models to use for a query. Current methods are mostly subjective or based on narrow benchmarks, limiting fair comparison and practical adoption.

2.2 Problem Statement

Users and researchers lack a scalable tool to evaluate responses to queries when asked by different models, and therefore cannot determine at times which models are best suited to respond to a given query, or a dataset.

2.3 Project Objectives

The goal of the project is to:

- Research and determine how we can build a system which decides on a 'winning' response given a query and multiple models.
- Allow users to input queries and receive outputs from multiple models.
- Implement LLM-as-a-judge cross-evaluation where models assess each other's answers.
- Implement human intervention/feedback for when the user disagrees with the system's assessment
- Provide a consensus "winner" and ranking of responses with a score-based grading of all responses
- Enable integration of new models through API keys.
- Evaluate the performance of our system through various metrics and situations built for testing LLM-as-a-judge systems through large scale experimentation

2.4 Proposed Solution

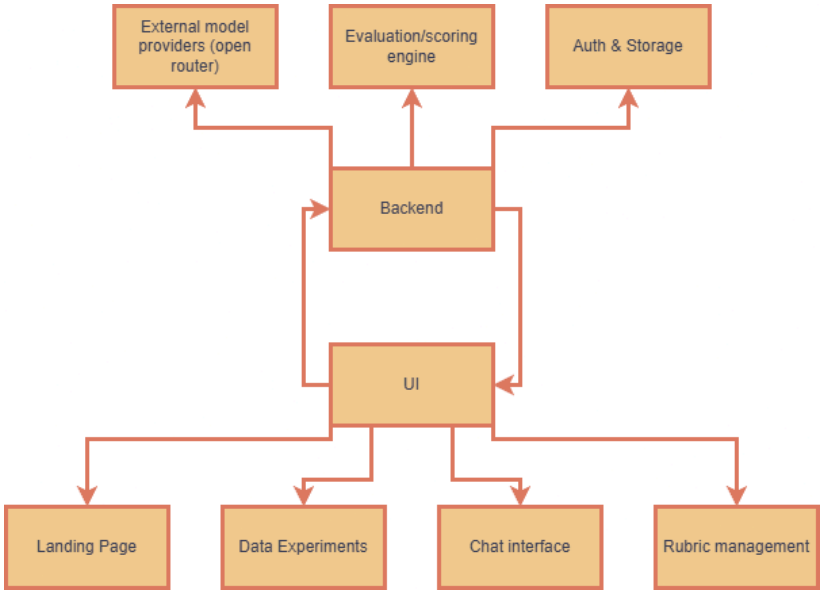
We will develop a web-based application that automates model response evaluation and presents results through a user-friendly interface. Using an LLM-as-a-judge based cross-model evaluation, the system will identify the 'best' response for a given query while remaining scalable and adaptable. We will also perform experiments and record our findings in a notebook to evaluate the effectiveness of our system.

3. Purpose of this Document

The purpose of this Design and Verification & Validation (V&V) document is to provide a detailed technical blueprint for the implementation and testing of the LM Compass platform. Specifically, the document aims to define a detailed breakdown of components and usage flow, establish a mapping of requirements from our SRS to the actual system, and outline the methodologies and metrics used to verify that the system functions correctly and as-intended.

4. Project Component Diagram

Below is a high level component diagram of our system. In section 5, we further break down these components based on how they have been structured in our codebase.



1. **Client (Frontend)**: Next.js 14 Application (React/Shadcn UI).
    - Landing page
    - Chat interface
    - Custom Rubric management
    - Experiments tab for large datasets
2. **Backend**: Next.js App Router API Routes (/api/chat).
    - Authentication and Storage: Supabase (Auth, PostgreSQL Database).
    - External Model Providers: 3rd Party APIs (OpenAI, Anthropic, Google and More through OpenRouter).
    - Evaluation/scoring Engine: Internal Library (lib/evaluation) for different LLM-as-a-judge evaluation methods.

5. Relationship between Components and Requirements

Below is a mapping of SRS requirements (including additions) to the specific codebase

components implementing them (or high level component plan for requirements not yet implemented).

Component	**Requirements (Based on SRS)**	**Description**
UI: Landing Page	P2: Implement landing page	Users encounter a public-facing page that introduces the tool, features marketing copy, an overview of capabilities, and a "Get Started" call to action.
UI: Data Experiments	P1: Data experiments and large scale evaluations	Provides a dashboard for users to import datasets of queries. This allows for batch evaluation runs across multiple models (*n* models) to test generalized output performance and suitability without manual entry for each prompt.
UI: Chat interface	P0: Submit Queries, Candidate Generation P1: User Input on Non-Consensus, User Input & Feedback on Responses P2: Share/Export Results, Multimodal Inputs, Model Selection	The core interaction layer. It allows users to submit free-text prompts, view streaming responses from multiple models simultaneously (Candidate Generation), and manually select preferred responses or up/down-vote answers. It also supports exporting results and selecting open-source models via dropdowns. Future iterations will include multimodal file attachments.
UI: Rubric Management	P1: CRUD Operations for custom Rubric management	Users define custom criteria for the judges.
Backend: External model providers (OpenRouter)	P0: Model API Key Vault & Routing, Candidate Generation (Multi-Model) P2: Model Selection from	Handles the logic for securely storing/retrieving API keys (via client-side encryption or secure storage) and routing user requests to the

	OpenRouter models	appropriate 3rd-party APIs (OpenAI, Anthropic, etc. through OpenRouter). It ensures queries are sent to all selected models concurrently.
Backend: Evaluation/scoring Engine	P0: Judging & Scoring Engine	The "LLM-as-a-judge" implementation. This backend service takes the user's prompt, the models' responses, and the selected rubric to calculate accuracy metrics and determine a "winner" or score for each response based on a selected evaluation method.
Backend: Auth & Storage	P1: Query and Result History P2: User Authentication System P3: User Feedback Mechanism	Manages user identity (Sign Up, Login, Session Management) and persistence. It stores chat history, evaluation results, user feedback (bug reports), and user preferences (like API keys or dark mode settings) in the database.

6. Component Behaviour

6.1 UI: Landing Page Component (to be implemented)

- **Normal Behaviour:** This component will serve as the public-facing front page of the LM Compass application. It will provide an entry point for users before authentication, display key information about the platform's capabilities, showcase key features, and guide users to sign in or get started with the application. The page should be accessible without authentication and provide a welcoming introduction to the peer-review LLM evaluation platform.

- **API and Structure (will be under *app/home/page.tsx*):** When a user navigates to the landing page, the component will check authentication status via Clerk. If authenticated, users will be redirected to the main application. Clicking "Sign In" or "Get Started" will trigger navigation to Clerk authentication or the main app. The page will render static content (hero, features, CTAs) without database interactions.

- **Implementation:** The technologies used are Next.js App Router, React, TypeScript, and

Clerk for authentication. When the landing page loads, it checks the user's authentication status via useUser() from Clerk. If authenticated, handleGetStarted() redirects to the main application. When the sign-in button is pressed, handleSignIn() opens Clerk's sign-in modal. The component uses Next.js Link for navigation and Tailwind CSS for responsive styling.

- **Potential Undesired Behaviours:** Authenticated users may see the landing page unnecessarily if authentication status is not checked on mount. Heavy assets could slow initial page load, requiring image optimization and lazy loading. The layout may break on small screens without responsive design.

6.2 UI: Data Experiments Component (to be implemented)

- **Normal Behaviour:** This component will enable users to import a dataset of queries to run batch evaluations across *n* models to test generalization and performance. Users will be able to upload CSV files containing queries, select multiple models for evaluation, choose evaluation methods, monitor progress of batch runs, and view analytics and results. The component will provide a dashboard for managing experiments, viewing historical batch runs, and analyzing performance metrics across models.

- **API and Structure (will be under *app/experiments/page.tsx*):** When a CSV file is uploaded, it is validated and parsed to extract queries. The parsed queries are sent to /api/experiments/batch with selected models and evaluation method. The API creates an experiment record in Supabase and queues queries for processing. The queue system processes queries by calling the Chat API for each query-model combination. Progress is tracked in the database and polled by the frontend. Upon completion, results are aggregated and analytics are computed and stored.

- **Implementation:** The technologies that will be used are Next.js, React, TypeScript and CSV parsing libraries. When a CSV file is uploaded, a function such as handleFileUpload will validate and parse the file, then display a preview. When submitted, a function like createExperiment will create an experiment record and queue queries. Progress will be tracked in the database and polled by a function like monitorProgress. Upon completion, we will fetch and display aggregated results.

- **Potential Undesired Behaviours:** Malformed CSV files could cause parsing failures without format validation. Too many concurrent queries could overwhelm the system without rate limiting. Processing many queries simultaneously could exhaust memory without batch size limits. Some queries might fail while others succeed, requiring continued processing and failure reports.

6.3 UI: Chat Interface Component

- **Normal Behaviour:** The Chat Interface should allow users to submit prompts, view streaming responses from multiple models (P0), manually select preferred responses or provide feedback (P1), and export results (P2). It displays messages chronologically, shows loading states, presents evaluation results, and handles tie scenarios.

- **API and Structure:** When a user submits a prompt, the query is sent to the Chat API with

selected models and evaluation methods. The API routes to OpenRouter, which queries all models concurrently. Responses stream back via SSE with querying, evaluating, and completion phases. All candidate responses are displayed simultaneously. For ties, users can select a winner, stored in message state and database. Feedback (P1) is sent to a feedback API and stored in Supabase. Export (P2) will format data as text, markdown, JSON, or CSV. Multimodal inputs (P2) will process file attachments and include them in query payloads.

- **Implementation:** The technologies used are React, TypeScript, Next.js API routes, Server-Sent Events, and Supabase. When a user submits a query, handleSubmit in PromptInputComponent sends a POST request to /api/chat. The API queries all models concurrently and streams responses via SSE. The frontend receives updates and renders ModelResponseCard components for each response. For ties, handleSelectWinner updates the message state. Feedback (P1) is stored via handleFeedback calling the feedback API endpoint. Export (P2) will use formatting functions like exportAsMarkdown to transform data and trigger downloads. Multimodal inputs (P2) will process files and convert them to base64 before including in API requests.

- **Potential Undesired Behaviours:** Network failures during SSE streaming could leave the UI inconsistent without error boundaries and retry mechanisms. Network errors could prevent feedback from being saved without local queuing. Large responses might cause export failures without streaming generation.

6.4 UI: Rubric Management Component

- **Normal Behaviour:** The Rubric Management component provides CRUD operations for users to define and manage custom evaluation criteria. Users can create, view, edit, delete, and select rubrics for use in evaluations.

- **API and Structure:** When creating a rubric, the name and description are sent to server actions, validated, and stored in the Supabase rubrics table with user ID and timestamps. Retrieving rubrics queries the database filtered by user ID, optionally including system defaults. Updates modify existing records, while deletions check for active usage first. When selected for evaluation, the rubric ID is passed to the evaluation system, which loads the content to construct scoring prompts.

- **Implementation:** The technologies used are React, TypeScript, Next.js Server Actions, and Supabase. When the page loads, loadRubrics calls getUserRubrics to fetch rubrics from Supabase. Rubrics are displayed in a list view. Creating a rubric opens AddRubricDialog, and handleSave calls the createRubric server action which authenticates the user and inserts the rubric. Editing calls updateRubric to update the database record. Deleting checks for active usage before calling the delete server action. The selected rubric is stored in state and passed to the evaluation system.

- **Potential Undesired Behaviours:** Very long descriptions could cause performance issues without size limits. Users might edit rubrics they don't own without ownership verification. Updating rubrics used in past evaluations could affect historical data without versioning.

6.5 Backend: External model providers (OpenRouter) Component

- **Normal Behavior:** The External Model Providers component handles secure API key storage, request routing to OpenRouter, and concurrent querying of multiple models. It provides a unified interface for interacting with multiple LLM providers through OpenRouter's API.

- **API and Structure:** When a user provides an OpenRouter API key, it is encrypted using a strong encryption algorithm (see lib/encryption.ts) and stored in the Supabase user_settings table. For candidate generation, the system retrieves and decrypts the key, creates an OpenAI-compatible client with OpenRouter's endpoint, and sends queries to all selected models concurrently. OpenRouter routes requests to appropriate providers and returns responses with usage metadata. For dynamic model selection (P2), the system fetches available models from OpenRouter's API, filters by capabilities and pricing, and caches results with periodic syncing.

- **Implementation:** The technologies used are Node.js, the OpenAI SDK (configured for OpenRouter), the encryption module, and Supabase. When saving an API key, saveOpenRouterKey encrypts the key using the encryption module before storing it in Supabase. When querying models, retrieveApiKey fetches and decrypts the key, then creates an OpenAI client with OpenRouter's base URL. The routeToOpenRouter method uses Promise.allSettled to send concurrent requests to all selected models. For dynamic model selection (P2), fetchOpenRouterModels calls OpenRouter's models API, filters results, and caches them. The cache is refreshed periodically by syncModelList.

- **Potential Undesired Behaviors:** Stolen API keys could allow unauthorized usage without strong encryption. External API downtime could break generation without retry logic. Rate limits could throttle requests without queuing. Models might become unavailable without availability validation. Uncontrolled usage could lead to high costs without usage tracking. Too many parallel requests could overwhelm the system without concurrency limits. Cached model lists might not reflect new models without cache invalidation. Invalid keys could cause authentication errors without validation. Slow responses could cause timeouts without configurable timeouts.

6.6 Backend: Evaluation/Scoring Engine Component

- **Normal Behaviour:** The Evaluation/Scoring Engine is the "LLM-as-a-judge" implementation. This backend service takes the user's prompt, the models' responses, and the selected rubric to calculate accuracy metrics and determine a "winner" or score for each response based on a selected evaluation method. It supports multiple evaluation methods including prompt-based (n^2 approach) and n-prompt-based approaches, where judge models evaluate candidate responses against rubric criteria.

- **API and Structure:** When evaluation is triggered, the service receives the user's query, multiple model responses, and the selected rubric. The system loads the rubric content from the filesystem or database, then constructs scoring prompts for each judge-evaluated pair (or single prompt per judge for n-prompt method). These prompts are sent to judge models via

OpenRouter, which return JSON-formatted scores and reasoning. The system extracts scores from the responses, calculates mean scores for each evaluated model, and determines the winner or identifies ties. Evaluation results including scores, reasoning, and winner information are returned to the calling service and stored with the message metadata.

- **Implementation:** The technologies used are Node.js, TypeScript, the OpenAI SDK (configured for OpenRouter), and filesystem access for rubric loading. When the Chat API route handler calls the evaluator's evaluate method, it passes the model responses and evaluation options. The PromptBasedEvaluator creates n^2 scoring queries where each model judges all other models, while NPromptBasedEvaluator creates n queries where each model judges all others in a single prompt. The buildScoringQueries method constructs prompts with the user query, candidate responses, and rubric. All evaluation queries are sent concurrently using Promise.allSettled. The extractScoreAndReasoning method parses JSON from responses, handling markdown code blocks and brace counting for robust extraction. The calculateMeanScores method aggregates scores per model, and findWinner determines the winner or identifies tied models.

- **Potential Undesired Behaviours:** Judge models might return malformed JSON or non-JSON text, requiring us to have null score assignment. The rubric file might be missing or inaccessible, requiring graceful fallback to empty rubric. Some judge models might fail to respond or timeout, requiring continued processing with available evaluations. JSON structure might not match expected format, requiring type validation and default null scores. Different judge models might interpret rubric differently, requiring clear instructions and explicit scoring guidelines.

6.7 Backend: Auth & Storage Component

- **Normal Behaviour:** The Auth & Storage backend component manages user identity (Sign Up, Login, Session Management) and persistence. It stores chat history, evaluation results, user feedback (bug reports), and user preferences (like API keys or dark mode settings) in the database. The component handles authentication through Clerk, manages user sessions, persists chat conversations with only winning responses, stores user settings securely, and manages feedback submissions.

- **API and Structure:** When a user signs up or logs in, Clerk handles authentication and provides a user session. The system stores user identity information and associates all user data with the user ID. Chat history is stored in Supabase with chat records containing user ID, title, and timestamps, while messages are stored with chat ID, role, content, metadata (including evaluation results), and sequence order. Only the winning response is stored in chat history for each evaluation. User settings including encrypted API keys are stored in the user_settings table. User feedback and bug reports are stored in a feedback table with user ID, message content, and timestamps. The system retrieves chat history on demand, loads messages with pagination support, and manages user preferences across sessions.

- **Implementation:** The technologies used are Clerk for authentication, Supabase for database storage, Next.js Server Actions, and the encryption module for sensitive data. When a user

authenticates, Clerk provides session management and user identity. The Chat Context Provider manages chat state and calls storage functions to persist and retrieve data. The saveChat function upserts chat records and inserts messages with sequence ordering. The loadChat function retrieves messages with pagination, loading the last N messages and supporting infinite scroll. The listChats function queries all user chats ordered by update time. User settings are stored via server actions that encrypt sensitive data before storage. Feedback submissions are handled through server actions that insert records into the feedback table. The system uses Row Level Security policies in Supabase to ensure users can only access their own data.

- **Potential Undesired Behaviours:** Concurrent saves could overwrite each other without database transactions. Sequence order gaps could occur from deleted messages without re-sequencing. Ownership verification might fail without proper authentication checks. Large message arrays could exceed database limits without batching. User settings might not persist without proper error handling.

7. User Interface Design

7.1 Design Philosophy & Visuals

The application adopts a "Content-First" minimalist design philosophy, specifically tailored for the text-heavy nature of Large Language Model (LLM) evaluation. To ensure distraction-free comparison, subtle accents are employed strategically to guide user attention to critical features (such as alerts or login call to actions) without overwhelming the primary content. The application uses a clean, modern Sans-Serif font family (Poppins), which have higher legibility at various sizes. Monospaced fonts are also used specifically for code blocks within model responses.

7.2 Layout & Navigation

The screen is divided into three functional zones designed for flexibility:

- **Sidebar:** A collapsible and resizable navigation pane, allowing users to navigate through our app while minimizing screen real estate.
- **Main Stage**: The central interface that houses the chat messages and the comparison panel, serving as the primary workspace for viewing model outputs.
- **Top Navigation Bar**: Located above the stage, this bar provides quick access to important high-level controls, allowing users to adjust model parameters, switch between rubric modes, and toggle dark mode.

This layout strategy positions navigation and configuration controls at the periphery to dedicate the maximum possible screen real estate to the main chat and comparison panel.

7.3 User Interactions

The interaction workflow is designed for clarity and control. Users begin by defining comparison targets via the Multi-Model Selector and configuring criteria in the Rubric Manager. Upon

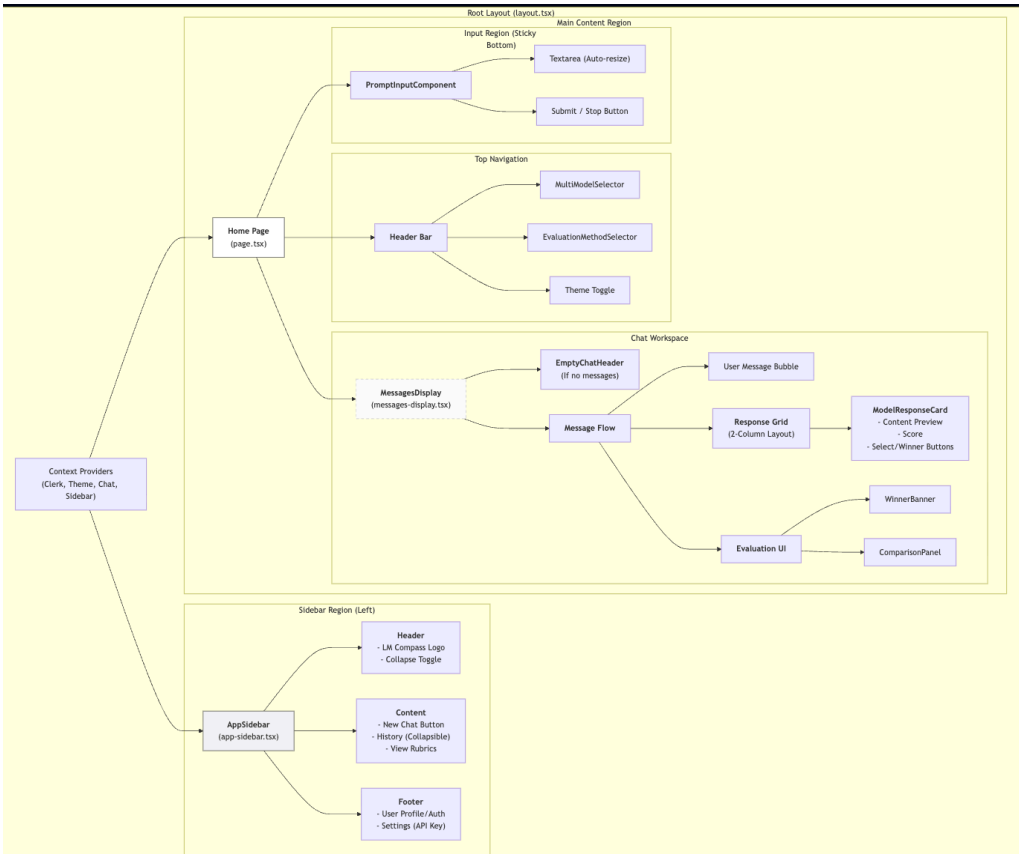
prompt submission, the interface enters a dynamic "thinking" state, featuring a dancing dots animation and status updates (e.g., "Querying 2 models," "Evaluating responses") to visualize backend progress. Once complete, users review the side-by-side outputs in the comparison panel and see the resulting scores and analysis based on the rubric.

7.4 Planned Interface Expansions

Landing & Onboarding: The future Landing Page will serve as the app's entry point, highlighting key value propositions. Users will interact with a prominent call-to-action (e.g., "Start Evaluation") to seamlessly transition into the authentication flow or directly into the comparison workspace.

Rubrics Page: Our existing rubrics page will be updated to house an interface for granular control over evaluation criteria. Users will interact with a structured list view to manage custom rubrics, utilizing modal-based editors to define specific scoring parameters, weights, and descriptions, ensuring consistent judging standards across sessions.

Experiments Dashboard: To support bulk analysis, a new Experiments Tab will allow users to upload external datasets (e.g., CSV/JSON) for batch processing. This interface is designed for asynchronous interaction; users will initiate long-running evaluations, track status via progress bars, and ultimately view a Results Visualization pane populated with comparative charts and performance metrics once the experiment concludes.



This schema chart guided our UI implementation by enforcing a clean separation of concerns, dividing

the screen into three distinct functional zones.

8. Component Test Plan (V&V)

8.1 Unit Tests

Unit tests will verify desired behaviour and lack of undesired behaviours for all components of the LM Compass system. Tests will be implemented using Jest or Vitest, with React Testing Library for frontend components and standard testing utilities for backend components.

8.1.1 Front End Components

PromptInput: Unit tests for the PromptInput component will verify that user input handling and submission logic function correctly with proper validation and error handling. These tests will show that input validation prevents invalid submissions, valid submissions trigger API calls with correct data, and that various API error conditions are handled appropriately with user-friendly messages. Request cancellation behaviour will be tested to ensure that loading states are properly managed throughout the request lifecycle. Streaming response handling will be tested to ensure content is displayed to the user as it arrives.

SettingsDialog: Unit tests for the SettingsDialog component will ensure that API key input validation and form handling function correctly. Tests will verify that invalid API key formats are detected and appropriate error messages are displayed, that valid submissions trigger save operations correctly, and that success and failure scenarios are handled appropriately with proper UI feedback. Further unit tests will be implemented to confirm that form state is managed correctly, including loading states and input clearing behaviours.

8.1.2 Back End Components

Encryption Module: Unit tests for the encryption module will confirm that encryption and decryption functions operate correctly. Tests will verify that invalid or corrupted ciphertext is handled with appropriate error handling. The module will be tested to ensure it validates encryption key configuration at initialization and rejects invalid configurations.

Settings Actions: Unit tests for the settings actions module will verify that API key management functions handle authentication, encryption, and database operations correctly. Tests will prove that API keys are properly encrypted before storage. Authentication checks will be tested to ensure unauthorized requests are rejected and authorized requests proceed correctly.

Chat API Route: Unit tests for the chat API route will verify that the endpoint handles authentication, authorization, and request processing correctly. Tests will validate that unauthenticated requests are properly rejected and that missing API keys are handled appropriately. Request validation tests will ensure missing or invalid model selections are rejected with appropriate error responses. Tests will prove that technical API errors from external services are explained in user-friendly error messages. The endpoint will be tested to guarantee streaming responses are properly formatted and that errors during evaluation are handled without crashing the server.

Evaluation Module: Unit tests for the evaluation module will verify that model response evaluation logic functions correctly under various conditions. Tests will affirm handling of single response scenarios and correct generation of scoring queries with all required components. The

module will be tested to assure it correctly parses evaluation responses from judge models, handles malformed inputs properly, and aggregates scores correctly. Winner determination logic will be tested to ensure that winning models are correctly identified and tie scenarios are handled appropriately.

Rubric Actions: Unit tests for the rubric actions module will show that rubric creation and retrieval operations work correctly with proper authentication and validation. Tests will confirm that required field validation is enforced, that authentication is properly checked for all operations, and that database errors are handled appropriately. The module will be tested to ensure successful operations return correct data structures and that error conditions are properly communicated to callers.

Data Experiments Actions: Unit tests for the data experiments actions module will verify the handling of batch experiment creation and management. Tests will confirm that dataset parsing correctly extracts queries. The module will be tested to ensure that experiment creation correctly inserts status and results of the experiment into the database. Tests will also cover the logic for queuing and processing individual queries within a batch, ensuring that the system can handle partial failures without stopping the entire experiment.

8.2 Performance Tests and Metrics

Response Accuracy: The LLM-based judging system shall be evaluated against a human-annotated evaluation dataset serving as the ground truth. This dataset will primarily consist of questions & answers from the Humanities Last Exam dataset. Evaluator performance will be measured by agreement with human judgments (e.g., rank correlation or inter-rater agreement).

Speed/latency: The system shall meet acceptable response-time limits for evaluation requests. End-to-end latency will be measured as a function of the number of models evaluated, and limits on the number of models used will be placed accordingly. Ideally, when executed asynchronously, total evaluation time is expected to be approximately bounded by the slowest model response, however, the use of OpenRouter as a hub for all LLM queries slows down response time, likely due to an OpenRouter resource cap.