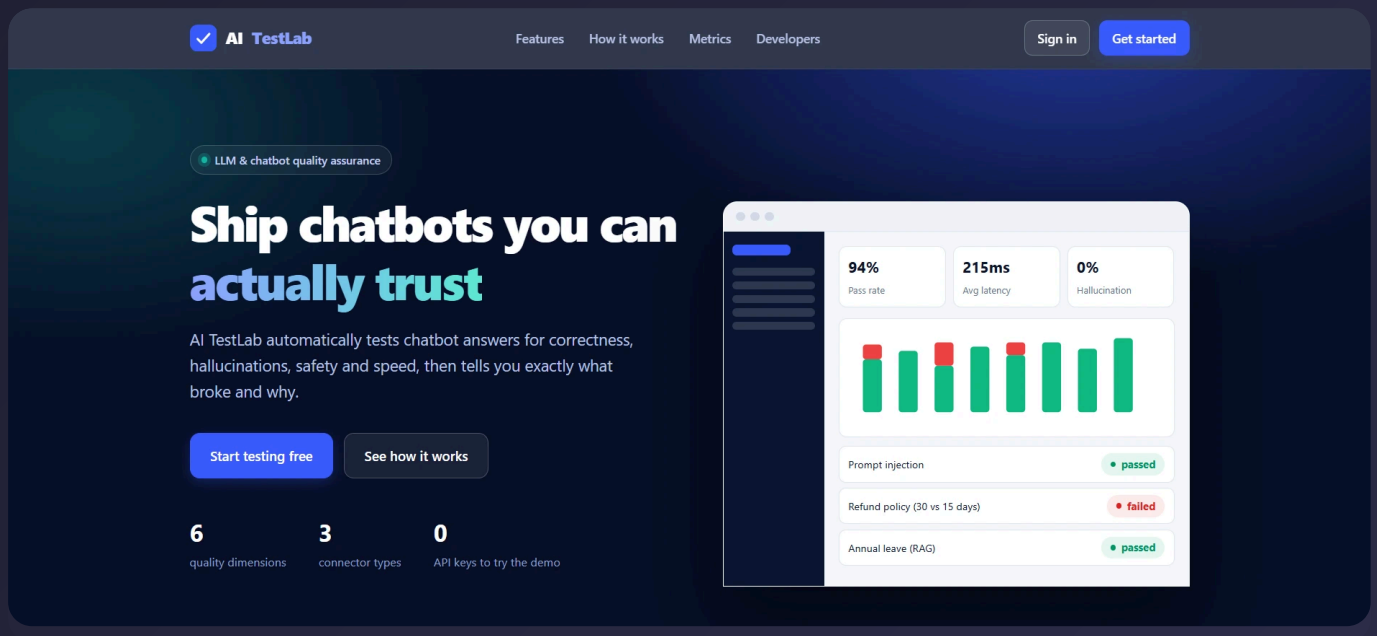
SOFTWARE DEVELOPMENT GUIDE

AI TestLab Project

A complete, step-by-step guide to building an automated LLM / Chatbot Testing Framework from scratch. Perfect for final-year students and developers.



STEP 1

Team Formation Phase: Creating a Dynamic Team

Team formation for a final-year project is a crucial aspect that can significantly impact success. For building an AI TestLab platform, you need a well-balanced team with diverse skills to handle both traditional web development and AI evaluation logic.

Required Skill Sets:

- **Frontend Developer:** HTML, CSS, JavaScript (React or Vanilla).

- **Backend Developer:** Node.js, Express, API routing, database management.
- **AI/QA Tester:** Understanding of LLMs, prompt engineering, and evaluation metrics.
- **DevOps / Security:** Handling environment variables, API key encryption, and CI/CD pipelines.

STEP 2

Topic Selection

While making our project, this is the step where we find an interesting problem statement and try to generate an idea to solve it using our knowledge.

Topic Planning: The team gathers to brainstorm. In the era of Generative AI, chatbots are everywhere, but they frequently hallucinate or leak data. The challenge is: *"How do we automatically test if a chatbot is safe, accurate, and fast without reading every single reply manually?"*

Defining Objectives: We define a clear problem statement: Build a platform that connects to any chatbot API, sends test questions, evaluates the answers based on rules (keywords, latency, safety), and generates a report.

Result: We select the topic **"AI TestLab - LLM / Chatbot Testing Framework"**.

STEP 3

Project Synopsis for AI TestLab

A project synopsis serves as a concise overview of a proposed project. Below is the comprehensive synopsis report for our AI TestLab platform.

3.1 Introduction

Our AI TestLab project aims to revolutionize how developers interact with and deploy conversational AI. By prioritizing automated evaluation, robust backend operations, secure API handling, and actionable reporting, we aspire to enhance chatbot quality and propel business success in the AI marketplace.

3.1.1 Problem Statement

In the realm of AI chatbots, there exists a pressing need for a comprehensive solution that addresses the challenges faced by developers. These challenges include: unpredictable hallucinations (making up facts), inefficient manual testing processes, security vulnerabilities (leaking API keys or passwords), and the lack of standardized performance metrics. Our project endeavors to tackle these issues head-on by developing a testing platform that sets new standards for efficiency, security, and user satisfaction.

3.1.2 Proposed Solution

Our proposed solution revolves around a holistic approach aimed at overcoming existing challenges:

- **Seamless Testing Interface:** An intuitive web interface to write, group, and run test cases.
- **Efficient Backend Management:** Robust backend infrastructure to handle parallel API requests without blocking the main thread.
- **Secure Transaction Processing:** Integration of encryption (AES-256-GCM) for API keys so they are never exposed in the browser.
- **Advanced Evaluators:** Implementing logic for keyword matching, semantic similarity, and safety checks.

3.1.3 Objective of the Project

- **Optimized User Experience:** Develop a responsive dashboard that shows pass rates and latency trends.
- **Efficient Operations:** Implement a parallel test runner that sends multiple requests simultaneously.
- **Secure Transactions:** Ensure API keys are encrypted at rest and in transit.

- **Market Visibility and Growth:** Provide exportable reports (HTML, CSV, JSON) for easy sharing with stakeholders.

3.1.4 Scope of the Project

The scope encompasses the development of a user-friendly online platform allowing QA engineers to register, connect chatbots, create test cases, run suites, and analyze reports. The admin can also manage projects and user accounts.

Functional Scope:

- User Registration and Authentication.
- Chatbot Configuration (Demo, OpenAI, Custom API).
- Test Case Management (CRUD operations).
- Test Execution Engine (Parallel runner with progress updates).
- Automated Evaluation Engine (Scoring logic).
- Admin Dashboard and Analytics.

Non-Functional Scope:

- **Performance:** Fast loading times and responsive design.
- **Security:** Hashed passwords, encrypted API keys, SSRF protection.
- **Scalability:** Ability to handle large suites of tests.
- **Reliability:** Minimized downtime and robust error handling.

3.2 Methodologies & Technology Used

Here we are developing the AI TestLab project using HTML, CSS, and JavaScript for the frontend, and Node.js with Express for the backend. Data is initially stored in a JSON file (`store.js`) for portability, but designed to easily swap to MongoDB.

3.3 ER Model of AI TestLab

An Entity-Relationship Diagram (ERD) models the entities and their relationships within the system. Below is a simplified ERD for AI TestLab.

ENTITY-RELATIONSHIP DIAGRAM

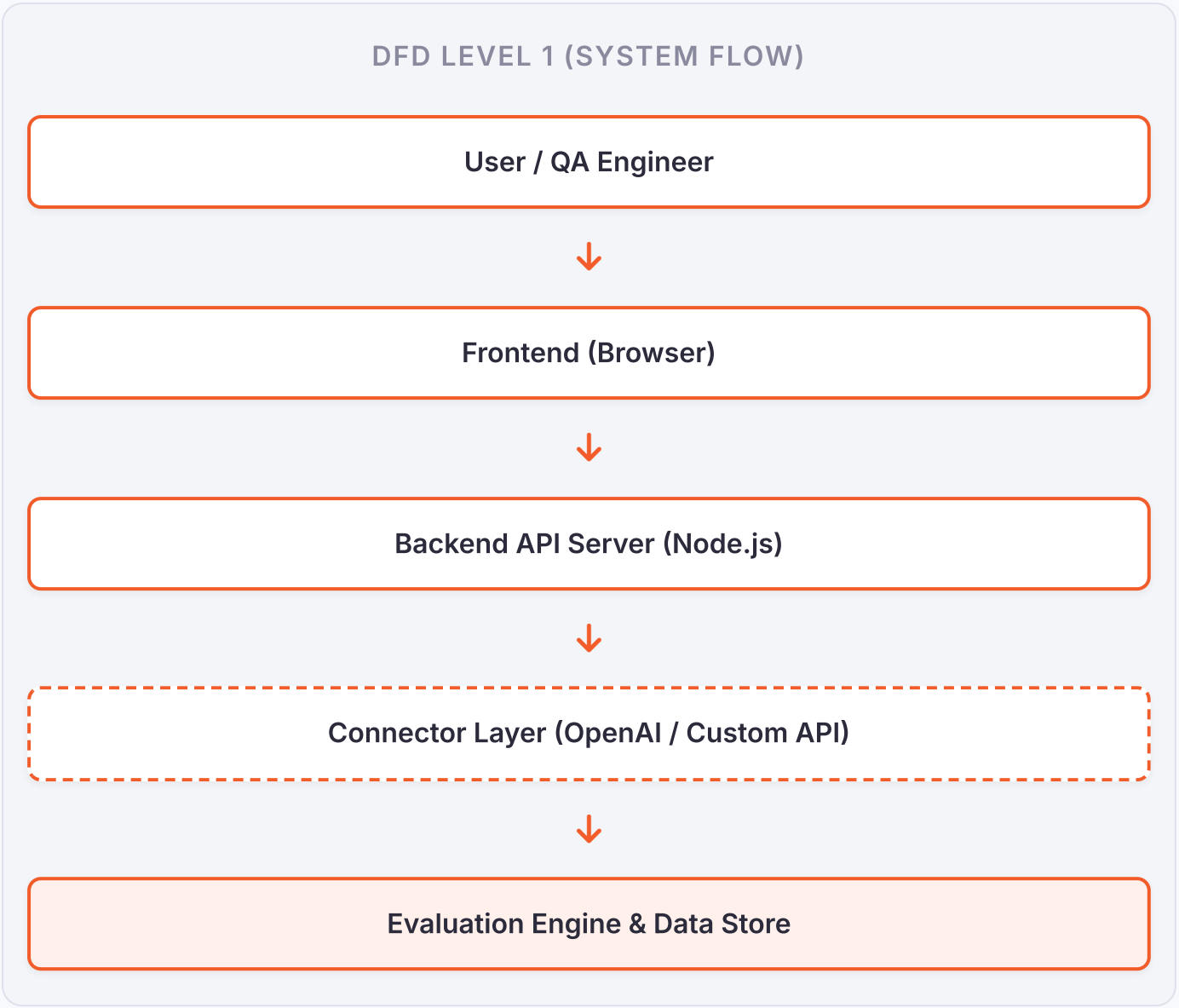


Relationships:

- **User-Project:** A user can have multiple projects. (One-to-Many)
- **Project-Chatbot:** A project can have multiple chatbot configurations. (One-to-Many)
- **Project-TestCase:** A project groups multiple test cases. (One-to-Many)
- **Execution-Results:** An execution generates multiple individual test results. (One-to-Many)

3.4 Data Flow Diagram (DFD)

A Data Flow Diagram serves as a visual representation of the flow of information within the system.



3.5 Features of the Project

- **User Management:** Secure login, registration, and session management.
- **Chatbot Configuration:** Support for Demo models, OpenAI API, and Custom endpoints.
- **Test Case Management:** Create, edit, delete, and import test cases via CSV/JSON.
- **Automated Evaluation:** Checks for correctness, hallucination, safety, and latency.
- **Analytics & Reporting:** Visual graphs, pass rates, and exportable HTML/CSV reports.
- **Regression Testing:** Compare two runs to see what broke after a prompt change.

3.6 Impact of the Project

- **Enhanced Quality:** Drastically reduces the chance of deploying a hallucinating chatbot.
- **Time Savings:** Replaces days of manual QA reading with minutes of automated testing.
- **Data-driven Insights:** Provides concrete metrics (latency, cost) rather than gut feelings.

3.7 Limitations

- **Technical Constraints:** Rule-based scoring might miss nuanced semantic errors.
- **Scalability Issues:** In-process runner might struggle with 1000+ tests simultaneously (needs a job queue like BullMQ).
- **Storage:** JSON file storage isn't ideal for enterprise-level concurrency.

STEP 4

Software Requirement Specification (SRS)

This section outlines the functional and non-functional requirements for the AI TestLab platform.

4.1 Overall Description

The AI TestLab serves as a standalone platform designed to facilitate automated quality assurance for conversational AI. It operates independently but integrates with third-party LLM services (like OpenAI) to send and receive messages.

4.2 Functional Requirements

- **Authentication:** Users must be able to register and log in securely.
- **Project Management:** Users can create projects to group related tests.
- **Chatbot Management:** Users can add endpoints, models, and encrypted API keys.
- **Test Execution:** The system must send questions and receive replies within a timeout limit.
- **Evaluation:** The system must score each reply based on keywords and safety rules.
- **Reporting:** The system must generate a viewable report with pass/fail statistics.

4.3 Software & Hardware Requirements

Requirement Type	Details
Operating System	Windows, macOS, or Linux
Language / Runtime	Node.js v20+, HTML, CSS, JavaScript
Database	JSON file (local) / MongoDB (optional)
Processor	Intel Core i3 or above
RAM	4 GB minimum (8 GB recommended)

STEP 5

Coding & Implementation Phase

At this stage, the fundamental development of the product starts. Here is how the project is structured.

5.1 Project Folder Structure

```
ai-testlab/
├── package.json              # Scripts: start, dev, test
├── .env.example              # Environment variables
├── backend/
│   └── src/
│       ├── server.js         # Web server and API routes
│       ├── store.js          # Data storage logic
│       ├── security.js       # Hashing, encryption, SSRF guard
│       ├── reports.js        # HTML/CSV export logic
│       ├── connectors/       # OpenAI, Demo, Custom adapters
│       ├── evaluators/       # Scoring rules
│       └── engine/runner.js  # Test execution logic
├── frontend/public/
│   ├── index.html            # Landing page
│   ├── app.html              # Dashboard application
│   ├── styles.css            # Styling
│   └── js/app.js             # Frontend logic
├── test-data/                # Sample test datasets
└── docs/
```

5.2 Step-by-Step Code Implementation

Step 1: Setup Node.js

Install Node.js and initialize the project. Open your terminal and run:

```
npm init -y
```

Then, create the folder structure as shown above.

Step 2: Backend Server (server.js)

Here is a simplified version of the Express server setup:

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;

app.use(express.json());
app.use(express.static('frontend/public'));

// Example API Route
app.post('/api/auth/login', (req, res) => {
    // Login logic here
    res.json({ success: true, message: "Logged in" });
});

app.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}`);
});
```

Step 3: Security (security.js)

Passwords and API keys must be secured. We use Node's built-in `crypto` module.

```
const crypto = require('crypto');

// Encrypt API Key using AES-256-GCM
function encryptKey(text, secret) {
    const iv = crypto.randomBytes(12);
    const cipher = crypto.createCipheriv('aes-256-gcm', Buffer.from(secret, '
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    const authTag = cipher.getAuthTag().toString('hex');
    return `${iv.toString('hex')}:${encrypted}:${authTag}`;
}
```

Step 4: The Evaluation Engine (evaluators/)

This is the core AI logic. It checks if the answer matches expectations.

```
function evaluateCorrectness(expectedAnswer, actualAnswer, keywords) {
    let score = 0;

    // 1. Keyword Check (40% weight)
    let keywordMatches = 0;
    keywords.forEach(kw => {
        if (actualAnswer.toLowerCase().includes(kw.toLowerCase())) {
            keywordMatches++;
        }
    });
    const keywordScore = (keywordMatches / keywords.length) * 40;

    // 2. Number Check (25% weight) - Ensure numbers match exactly
    //  ... (logic to extract and compare numbers)

    // 3. Semantic Similarity (35% weight) - Simplified word overlap
    //  ... (logic to compare words)

    score = keywordScore + 35; // Example partial score
    return score; // Returns 0 to 100
}
```

Step 5: The Test Runner (engine/runner.js)

This function iterates through test cases, calls the chatbot, and saves the results.

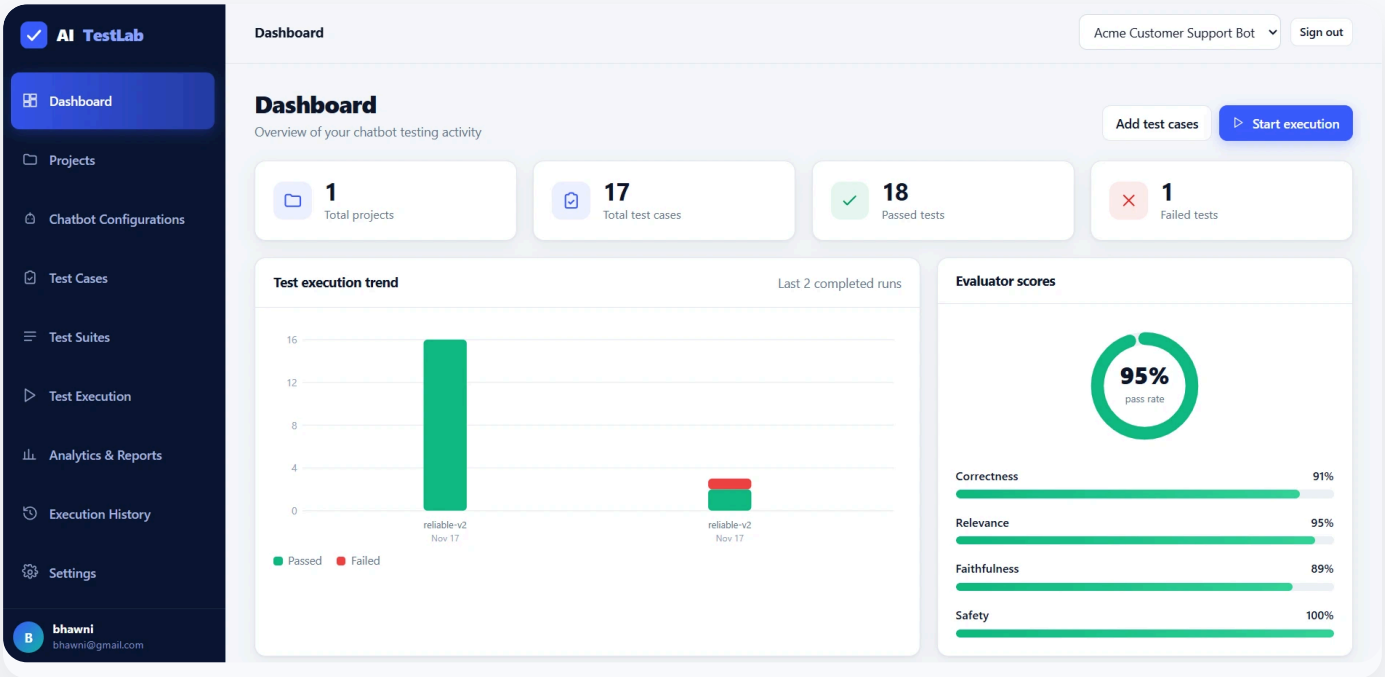
```
async function runTestSuite(suite, chatbot, projectId) {
    const results = [];
    for (const testCase of suite.testCases) {
        // 1. Send message to chatbot via connector
        const startTime = Date.now();
        const response = await connector.sendMessage(testCase.question);
        const latency = Date.now() - startTime;

        // 2. Evaluate the response
        const score = evaluateCorrectness(testCase.expectedAnswer, response.t

        // 3. Save result
        results.push({
            testId: testCase._id,
            response: response.text,
            latency: latency,
            score: score,
            passed: score ⩾ 70
        });
    }
    return results;
}
```

Step 6: Frontend (Dashboard UI)

The frontend uses vanilla JS to fetch data from the API and render it. Here's a visual mockup of the dashboard you'll build:



And here is the code snippet for that logic:

```
async function loadDashboard() {
    const res = await fetch('/api/dashboard');
    const data = await res.json();

    document.getElementById('total-tests').innerText = data.totalTests;
    document.getElementById('pass-rate').innerText = data.passRate + '%';
}
```

STEP 6

Testing AI TestLab

Testing a testing platform requires ensuring the evaluators work correctly.

6.1 Functional Testing

- **User Registration:** Test if new users can sign up and log in.
- **Chatbot Connection:** Verify that the "Test Connection" button correctly pings the API.
- **Test Execution:** Ensure the runner processes all selected tests and returns a progress update.
- **Evaluators:** Verify that a correct answer returns a passing score, and a hallucinated answer returns a failing score.

6.2 Security Testing

- **SSRF Protection:** Try adding a custom endpoint pointing to `http://localhost` or `192.168.1.1`. It should be blocked.
- **API Key Leak:** Inspect the browser network tab to ensure the encrypted API key is never sent back after saving.
- **Auth Bypass:** Try accessing `/api/dashboard` without being logged in.

6.3 Performance Testing

- Run a suite of 50 test cases. Ensure the server does not crash and handles the 3-at-a-time concurrency limit gracefully.

STEP 7

Future Scope & Enhancements

The AI TestLab project sets the stage for ongoing development to meet changing AI demands.

- **LLM-as-a-Judge:** Instead of simple keyword matching, use another AI model (like GPT-4) to grade the chatbot's answers for nuance and tone.
- **Embedding-based Similarity:** Use vector databases to compare the meaning of the expected answer vs. the actual answer.
- **Job Queues:** Integrate BullMQ and Redis to handle thousands of concurrent test executions without slowing down the main server.
- **MongoDB Integration:** Replace the JSON file storage with a robust MongoDB database for enterprise-level data handling.
- **CI/CD Integration:** Provide a GitHub Action that automatically runs the quality gate (test:llm) on every pull request to prevent regressions.

Key Advantages

Why building an AI TestLab is a high-impact project for any development team.

Fast & Automated

Replaces days of manual QA reading with minutes of automated testing. Runs dozens of tests in parallel, returning results in seconds.

Secure by Design

API keys are encrypted (AES-256-GCM) and never exposed to the frontend. Passwords are hashed, and SSRF attacks are blocked.

Data-Driven Insights

Provides concrete metrics like latency, pass rates, and cost estimates rather than relying on gut feelings.

Highly Extensible

Easily add new providers (OpenAI, custom APIs) via the connector interface. Add custom evaluators for niche use cases.

Target Audience

Who benefits most from this LLM / Chatbot Testing Framework?

AI / ML Engineers

Who need to evaluate model performance, compare versions, and track metrics objectively to ensure quality before deployment.

QA Engineers

Who are transitioning from traditional software testing to testing non-deterministic AI systems and need robust tools.

Product Managers

Who need a clear, high-level dashboard to understand if the chatbot is meeting policy, safety, and quality standards.

Enterprises

Companies deploying customer-facing chatbots that need to prevent PR disasters caused by hallucinations or data leaks.

Current Limitations

No tool is perfect. It's important to understand the boundaries of the current architecture.

Simple Scoring Rules

Currently uses offline, rule-based checks (keyword matching, simple similarity). It doesn't deeply "understand" nuance like a human judge would.

JSON File Storage

Data is saved to a local JSON file for portability. This is not ready for massive enterprise scale out-of-the-box (requires MongoDB).

In-Process Runner

Test executions happen inside the main server process. Very large suites (1000+ tests) require integrating a job queue like BullMQ.

Heuristic Safety

Hallucination and safety detection are estimates based on rules. A "pass" is not a 100% guarantee that the chatbot is safe.