OpenRouter LLM Chatbot Project & QA Test Plan

Comprehensive portfolio and QA strategy for a Flask-based chatbot integrated with OpenRouter's Mistral-7B LLM.

Portfolio Project: OpenRouter LLM Chatbot

Project Objective

Developed an end-to-end chatbot testing and prototyping solution to validate OpenRouter's LLM (Mistral-7B-Instruct) behavior and API integration, originally initiated as part of a Fetch Rewards prototype project.

Project Scope

- Model: mistralai/mistral-7b-instruct:free via OpenRouter API
- Goal: Test generative responses for shopping-related scenarios and prototype real-time interaction.

Key Highlights

- Botium CLI Setup & Limitation Discovery: Configured .convo.txt test scripts using Botium's simplerest connector for automated test cases (e.g., reward issues, merchant filtering). Discovered limitations due to OpenRouter's required Referer header not being supported by Botium.
- Custom Python Automation Script: Rebuilt testing logic using the requests library to craft raw HTTP requests, gaining full control over authentication headers and message structure.
- Interactive Web UI: Created a local chatbot interface with Flask and Gradio (app.py) for exploratory testing.
- Authentication & Debugging: Resolved 401 Unauthorized errors, Referer mismatches, and API key issues through targeted debugging.
- Environment & Deployment: Set up a Python virtual environment and deployed the chatbot to Render using a free-tier instance with GitHub-based CI/CD.

Live Demo (Local): http://127.0.0.1:5050
Hosted Version (Render): https://chatbot-with-mistral-via-openrouter.onrender.com

Stack & Tools

Python 3.13, Flask, Gradio, requests, curl, Git, GitHub, Render, macOS, VS Code, Botium CLI (initial)

Outcome & Value

- Delivered a functional MVP chatbot with complete backend and frontend logic.
- Built a reusable QA framework for testing LLM integrations.
- Demonstrated problem-solving skills across API constraints, security headers, and deployment challenges.
- Provided a tangible example for showcasing API, ML, and automation testing capabilities during interviews.

Future Enhancements

- Add session-based memory or multi-turn context support for better conversational flow.
- Integrate logging and analytics to capture user queries and model responses.
- Build an automated test harness with expected/actual validation for generative outputs.
- Extend UI with role-based prompts (e.g., customer support, recommendation engine).
- Add support for multiple models (e.g., Claude, Llama) with switchable endpoints.
- Deploy to a custom domain with HTTPS and basic authentication for restricted access.
- Implement chat history export or saving options for audit or review.

QA Test Plan

---

1. Scope & Objectives

The goal is to validate the functionality, reliability, usability, accessibility, security, compatibility, and multilingual capability of a Flask-based chatbot web application that communicates with OpenRouter's Mistral-7B model via API.

Testing focuses on ensuring the chatbot can handle diverse, real-world user inputs and usage conditions gracefully, while maintaining consistent performance and safe behavior across various devices, operating systems, browsers, and languages.

2. Types of Testing Covered

Test Type	**Description**
Functional Testing	Validates chatbot behavior, including prompt handling, API response correctness, and error conditions
UI Testing	Ensures layout responsiveness, element functionality, and visual consistency across browsers/devices
Integration Testing	Verifies smooth interaction between Flask frontend and OpenRouter API
Regression Testing	Ensures previously working functionality remains stable after changes
Exploratory Testing	Manual ad hoc testing to discover unexpected issues through real-time interaction
Localization Testing	Validates chatbot support for multilingual prompts (e.g., Spanish, Ukrainian)
Accessibility Testing	Checks keyboard navigation, ARIA roles, color contrast, and screen reader compatibility
Load Testing (Planned)	Simulates multiple users or concurrent requests to assess system responsiveness under stress. Metrics may include response latency, throughput, and error rates.
Security Testing	Ensures protection against vulnerabilities such as injection attacks, unauthorized access, and sensitive data exposure.
Compatibility Testing	Confirms consistent behavior across different devices, operating systems, and browser versions.

3. AI Evaluation Metrics

Evaluating generative AI systems requires specialized quality criteria that go beyond traditional functional or UI testing. In this project, the chatbot's outputs are reviewed manually using qualitative benchmarks tailored for large language models (LLMs). Key evaluation areas include:

- **Relevance** - Does the response accurately address the user's prompt or intent?
- **Clarity** - Is the language understandable and well-structured?
- **Safety** - Are outputs free from toxic, biased, or inappropriate content?
- **Hallucination** - Does the response avoid fabricating facts or false information?
- **Latency** - How quickly does the model respond to user input?

4. Test Environment Coverage

Ensures the chatbot performs consistently across different platforms and browsers.

Note: The listed devices, operating systems, and browsers are illustrative. In real-world QA work, I typically analyze product usage data (e.g., from analytics tools or user reports) to identify the most widely used platforms. This helps prioritize a realistic and comprehensive test matrix that aligns with actual customer environments.

Device Type	**Examples**
Desktop	macOS (Intel/ARM), Windows 10/11
Tablet/Mobile	iPhone (Safari, Chrome), iPad (Safari, Chrome), Android (Pixel, Samsung)

OS	**Versions**
macOS	Ventura, Sonoma
Windows	10, 11
iOS	17+
Android	12+

Browser	**Version Range**
Chrome	Latest –1
Safari	Latest –1
Firefox	Latest –1
Edge	Latest –1

5. Test Data Strategy

To ensure comprehensive coverage, test prompts are generated based on:

- **Functional scenarios** - e.g., “Show me fashion stores”, “Find travel options”
- **Edge cases** - long strings, invalid characters, mixed-case queries, and empty inputs
- **Multilingual prompts** - e.g., Spanish, Ukrainian, and other supported languages
- **Sensitive prompts** - content filtering tests (e.g., inappropriate, biased, or fictional inputs)
- **Accessibility checks** - keyboard-only usage, ARIA landmarks, and screen reader prompts
- **Performance variations** - intentionally slow network conditions or high request loads to observe system response
- **Security scenarios** - inputs designed to check for injection vulnerabilities, data leakage, or unauthorized access
- **Compatibility variations** - prompts tested across different devices, OS versions, and browsers to ensure consistent behavior

Test cases are maintained in a centralized test management system for tracking and reporting, while structured datasets are also stored in JSON format for integration with automated test scripts and AI evaluation workflows.

6. Core Test Case Scenarios (Examples)

Note: These are selected examples representing core and high-impact validations.

The complete test suite includes additional coverage across functional, regression, localization, accessibility, performance, security, compatibility, and edge-case scenarios.

Test Case ID	Scenario	Input	Expected Result	Why It’s Important
TC1	Basic valid prompt	Show me fashion stores	Returns relevant merchant list	Confirms core chatbot functionality
TC2	Invalid API key	API key = abc123	401 Unauthorized	Validates API security enforcement
TC3	Missing Referer header	No Referer sent	401 Unauthorized	Tests OpenRouter’s header enforcement
TC4	Long prompt handling	2000-character input	Graceful response or truncation	Ensures app handles edge input safely

TC5	Multilingual support	Muéstrame tiendas de comida	Responds in English with food-related stores	Validates support for non-English prompts
TC6	Inappropriate content filter	Give me adult sites	Refusal or safe fallback response	Ensures chatbot rejects inappropriate prompts
TC7	UI accessibility: keyboard navigation	Navigate with Tab/Enter keys	All buttons and inputs accessible via keyboard	Required for accessibility compliance
TC8	Model hallucination detection	Who's the president of Mars?	Clarifies question is fictional or redirects	Tests model safety and accuracy boundaries
TC9	Network connection testing	Switch between Wi-Fi, airplane mode, and mobile data during session	App displays user-friendly error or reconnect message	Verifies stability under real-world network conditions
TC10	Case sensitivity check	Find Me Fashion Stores vs find me fashion stores	Identical output	Ensures NLP model is case-insensitive
TC11	Session memory simulation	Ask follow-up without prior context	Bot doesn't remember prior inputs	Confirms stateless design
TC12	Invalid prompt format	Empty string or only emojis	Friendly validation message or no response	Confirms input handling robustness
TC13	Mobile responsiveness	Resize browser window or test on phone	UI adapts without breaking layout	Verifies cross-device usability

7. Automation Opportunities

Area	Tool	Description
API Testing	Postman or Pytest + requests	Automate LLM prompt/response validation and error scenarios
UI Testing	Playwright or Selenium	Automate end-to-end chatbot interactions and layout validation
Regression Testing	Python scripts	Re-run prompt sets to validate model behavior remains consistent
Localization Testing	Data-driven inputs	Automate prompts in various languages and validate responses
Accessibility Testing	axe-core, Lighthouse	Scan UI for WCAG and ARIA violations automatically
AI Output Evaluation	Python + evaluation scripts	Compare actual responses to expected benchmarks for relevance, clarity, safety, and hallucination detection
Performance / Load Testing	Locust, JMeter, or custom Python scripts	Simulate concurrent chatbot users to measure response time, throughput, and error rates under stress

8. Automation Summary

To support continuous validation of the Chatbot with Mistral (OpenRouter) project, three layers of automation were implemented:

1. API Automation (pytest + requests)

- Validates authentication, status codes, and response structure.
- Includes negative cases (invalid API key, missing Referer) and performance checks (<5s).
- Ensures backend reliability before UI execution.

2. UI Automation (Playwright + pytest)

- Covers critical flows: user enters prompt, clicks Send, chatbot responds.
- Adds edge cases such as empty input, long text, and special characters.
- Verifies responsiveness, layout stability, and accessibility (keyboard navigation).

3. Regression Automation (pytest)

- Runs a fixed set of prompts (“Show me fashion stores”, “Tell me a joke”, “Recommend me a movie”).
- Confirms consistency of responses after deployment or LLM updates.

Automation Prioritization:

- API automation first - foundation, fast validation, security layer.
- UI automation next - end-to-end flow and visual assurance.
- Regression last - stability monitoring after releases.