Architecture

Edit New page

Jump to bottom

Aditi edited this page 5 days ago · 11 revisions

Software Architecture Documentation

1. Overview

This system enables AI-driven software testing by automatically generating test code using Large Language Models (LLMs). It is designed to simplify and accelerate the creation of test cases for existing software, primarily Python codebases, with a focus on UI, logic, and persistence layer testing (though currently demonstrated mainly with unit tests for Python functions).

The solution is built for on-premise operation, leveraging local LLM runtimes like Ollama to ensure data security and control. Users can interact with the system, potentially through a chat-based interface (via the web frontend), to provide code and receive generated tests, aiming for a seamless and intuitive developer experience.

By utilizing LLMs managed by Ollama, the system can support incremental test code generation and aims to adapt to code changes. The long-term vision includes capabilities like creating self-healing scripts, ultimately reducing manual testing effort and increasing test coverage.

---

2. Key Components

The system is composed of several key components that work together:

1. **Frontend (Web Interface):**
    - **Description:** The primary user interface for the AI-Driven Testing system, developed as a **React/TypeScript Single Page Application (SPA)**. It enables users to input source code, formulate prompts (e.g., "Generate unit tests for this Python function"), potentially select a Large Language Model (LLM) if this feature is exposed, and view the test code generated by the backend. The frontend communicates with the Backend API via HTTP requests to submit tasks and retrieve results.

- **Location:** `frontend/` directory.
- **Key Files & Directories:**
  - `public/`: Contains static assets accessible by the browser.
    - `index.html`: The main HTML shell into which the React application is injected.
    - `manifest.json`: Web App Manifest enabling Progressive Web App (PWA) features.
    - `favicon.ico`, `robots.txt`: Standard web assets.
  - `src/`: This is the core directory containing all the application's source code, primarily React components written in TypeScript (`.ts`, `.tsx` files).
  - `package.json`: Defines project metadata, scripts (e.g., `npm start` for development, `npm run build` for production builds), and lists all Node.js dependencies (e.g., React, Material-UI, Emotion, TypeScript, `react-markdown`).
  - `package-lock.json`: Ensures reproducible installations of Node.js dependencies by locking their versions.
  - `tsconfig.json`: The TypeScript compiler configuration file, specifying how TypeScript code is checked and transpiled.
  - `Dockerfile`: Contains instructions to build a production-ready Docker image for the frontend. This typically involves building the static assets (using `npm run build`) and then serving them with a lightweight web server like `serve`.
  - `.gitignore`: Specifies files and directories (like `node_modules/`, `build/`) to be ignored by Git version control within the frontend subdirectory.

2. **Backend API (FastAPI Application):**

   - **Description:** A **Python-based API built with FastAPI**, running in a Docker container. It serves as the central hub, receiving requests from the Frontend. It orchestrates LLM interactions via the `LLMManager` and can apply pre/post-processing to prompts and responses using the `ModuleManager`.
   - **Location & Key Files:** `backend/api.py` (FastAPI app), `backend/schemas.py` (Pydantic models), `backend/Dockerfile`.

3. **LLM Orchestration Layer (`LLMManager`):**

   - **Description:** A core Python class located in `backend/llm_manager.py`. It is responsible for the entire lifecycle management of **Ollama Docker containers** (one per active model). This includes pulling the base `ollama/ollama` image, pulling specific LLM models (e.g., Mistral, Gemma) inside these containers, dynamically allocating free network ports, ensuring API readiness, sending processed prompts to the correct Ollama instance, and handling the streamed responses.
   - **Location & Key Files:** `backend/llm_manager.py`.

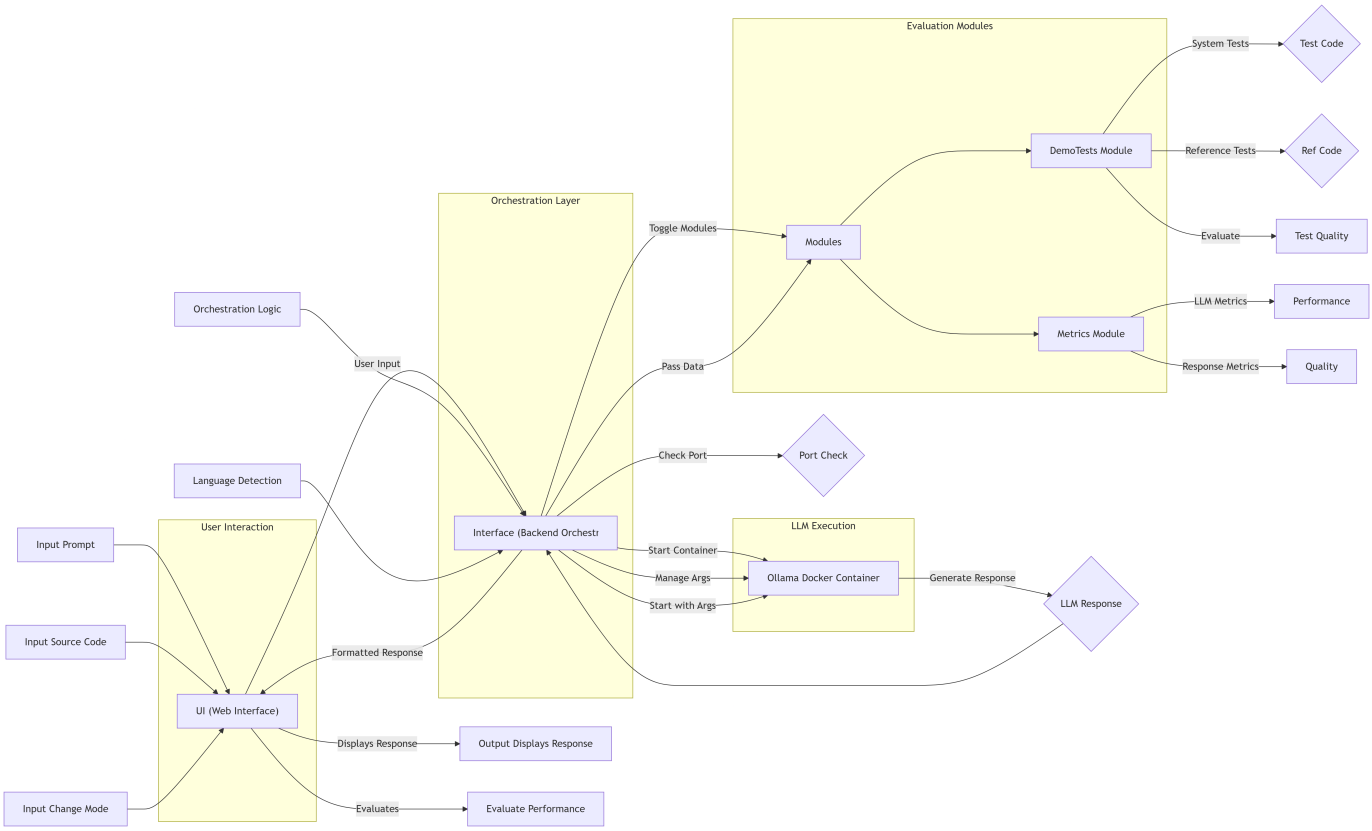
4. **Ollama Service (LLM Engine):**

- **Description:** The actual engine that runs the Large Language Models. The project uses **Ollama** (via the `ollama/ollama` Docker image) to serve various open-source LLMs locally. Each selected LLM runs in its own Ollama container, managed by the `LLMManager`.
- **Configuration:** The list of supported LLMs and their Ollama IDs is defined in `backend/allowed_models.json` and loaded via `backend/model_manager.py`. The actual model data is persisted in the `backend/ollama-models/` directory (mounted as a Docker volume).

5. **Test Generation Logic (LLMs & `ModuleManager`):**
   - **Description:** The core test code generation is performed by the **selected LLM** based on the (potentially pre-processed) prompt. The `ModuleManager` (defined in `backend/module_manager.py`) provides a plugin architecture. Custom Python modules (expected in a `backend/modules/` directory) can be invoked to pre-process `PromptData` before sending it to the LLM (e.g., for Retrieval Augmented Generation - RAG, as hinted by LangChain dependencies) and to post-process `ResponseData` from the LLM (e.g., extracting code blocks, validating syntax, formatting).
   - **Location & Key Files:** `backend/module_manager.py`, `backend/modules/` (presumed directory for custom modules).
6. **Output Handling & Storage:**
   - **Description:** LLM responses are primarily structured as Markdown. The backend, particularly through the flow orchestrated by `backend/execution.py`, saves the comprehensive `ResponseData` (which includes the Markdown output, model information, timing metrics, etc., as defined in `backend/schemas.py`) as structured JSON files. These are saved in both a timestamped archive (`outputs/archive/`) and a latest version (`outputs/latest/`). The Frontend is responsible for rendering the Markdown response to the user.
   - **Location & Key Files:** `backend/execution.py`, `backend/schemas.py`.
7. **Experimental Test Generation Framework (`ExampleTests/`):**
   - **Description:** A distinct sub-project within the repository focused on systematically generating and evaluating unit tests using various LLMs. It targets a curated set of Python programs found in `ExampleTests/pythonTestPrograms/` (both correct and faulty versions). This framework uses its own scripts (in `ExampleTests/Scripts/`) for automation and may use a simplified Ollama interaction mechanism or direct Docker commands. The results (generated test files) are stored in `ExampleTests/generatedTests/` for analysis and comparison of LLM performance.
   - **Location & Key Files:** `ExampleTests/` directory, `ExampleTests/README_tests` (methodology).

---

3. Architecture Diagram



(High-level system architecture showing user interaction with the frontend, which communicates with the backend API. The backend manages LLM operations via Ollama running in Docker containers.)

4. Technology Stack

Layer/Component	Technology / Tool	Purpose
Frontend	React, TypeScript, Node.js/npm, Material-UI, Emotion	User Interface, interaction with backend API
Backend API	Python 3.13+, FastAPI, Uvicorn	Web server, API endpoint definitions, request/response handling
LLM Orchestration	Python, `docker-py` (Python Docker SDK), `requests`	Managing Ollama Docker containers, model lifecycle, API communication

Layer/Component	Technology / Tool	Purpose
LLM Engine	Ollama, Docker	Running various open-source Large Language Models locally
Data Management (Backend)	Pydantic (for schemas), JSON	Structuring and validating data, storing LLM responses
Python Environment (Backend)	Conda	Managing Python dependencies for the backend
Test Generation (Core AI)	Various LLMs (e.g., Mistral, Gemma, DeepSeek, Qwen, Phi4 via Ollama - see `backend/allowed_models.json`)	Core AI-driven test code generation
Build/Orchestration (Overall)	Docker, Docker Compose	Containerization, multi-service application setup and management
Code Quality & Formatting	Black (formatter), Flake8 (linter), Pre-commit (git hooks)	Maintaining Python code standards
Testing Frameworks Used	`unittest` (for LLM-generated tests, reference tests in `ExampleTests`), `pytest` (for `ai-driven-testing/` app)	Automated testing of the project components and generated code
Version Control	Git, GitHub	Source code management and collaboration
CI/CD (DevOps)	Defined on CLI Pipeline page (e.g., GitHub Actions if used)	Automated build, test, and deployment pipeline
Advanced LLM Workflows (Hinted)	LangChain (`langchain`, `langchain-ollama`, `langchain-chroma` as per `backend/environment.yml`)	Potential for Retrieval Augmented Generation (RAG), complex chains, vector storage
`ai-driven-testing/` App Stack	Python, Flask, NumPy, Pandas (as per its `ai-driven-testing/requirements.txt`)	Specific technologies for the example/target application

5. Data Flow / Interaction Sequence (Typical Test Generation)

1. **User Interaction (Frontend):**
   - User navigates to the web UI (React app running on `http://localhost:3000`).
   - User inputs Python source code and a textual prompt (e.g., "Generate unit tests for this function.").
   - User might select a specific LLM from a list provided by the UI.
   - User submits the request via a button or action in the UI.
2. **Frontend to Backend API:**
   - The React frontend constructs an HTTP POST request.
   - The request is sent to the Backend API's `/prompt` endpoint (e.g., `http://backend:8000/prompt` when running via Docker Compose, or `http://localhost:8000/prompt` if backend is run directly).
   - The request body is a JSON object structured according to the `PromptData` Pydantic schema (defined in `backend/schemas.py`), containing the model ID, user message, source code, system message, and generation options.
3. **Backend API (`backend/api.py`):**
   - The FastAPI application receives the `PromptData` object.
   - The `/prompt` endpoint handler may pass this `PromptData` to the `ModuleManager` (`backend/module_manager.py`) for pre-processing if any "before" modules are active. This could modify the prompt, add context (e.g., via RAG), or adjust options.
   - The (potentially modified) `PromptData` containing the target `model_id` is then passed to the `LLMManager` instance.
4. **LLMManager (`backend/llm_manager.py`):**
   - The `start_model_container(model_id)` method is called (if the model's container isn't already active). This involves:
     - Verifying the `model_id` against `allowed_models.json`.
     - Pulling the base `ollama/ollama` Docker image (if not locally available).
     - Finding a free host port.
     - Starting a new Docker container for Ollama, configured to use the allocated port and mounting the `backend/ollama-models` volume for model persistence.
     - Waiting for the Ollama API within that new container to become responsive.
     - Instructing the Ollama instance (via its API) to pull the specific LLM (e.g., `mistral:7b...`) if it's not already present in the Ollama models volume.
   - The `send_prompt(prompt_data, ...)` method is called. It:

- Constructs the final prompt string (potentially using `rag_prompt` from `PromptData` or combining user message and source code).
- Prepares a JSON payload for the Ollama `/api/generate` endpoint, including the model ID, final prompt, system message, and generation options.
- Makes a streaming HTTP POST request to the specific Ollama container's API endpoint (e.g., `http://<ollama_container_name_or_localhost>:<port>/api/generate`).

5. **Ollama Service & LLM:**
   - The targeted Ollama container receives the generation request.
   - Ollama passes the prompt and options to the loaded LLM.
   - The LLM processes the input and generates the response (e.g., test code in Markdown format).
   - Ollama streams the generated tokens back as a series of JSON objects.
6. **LLMManager & Backend API (Response Handling):**
   - `LLMManager`'s `send_prompt` method collects the streamed JSON chunks, extracts the `response` content (which forms the Markdown text), and aggregates it.
   - It records timing metrics (loading time, generation time).
   - It constructs a `ResponseData` Pydantic object containing the original model metadata, the LLM's Markdown output, and the timing data.
   - This `ResponseData` object is returned to the `api.py` endpoint handler.
   - The endpoint handler may then pass this `ResponseData` (and the original `PromptData`) to the `ModuleManager` for post-processing if any "after" modules are active. These modules might extract code from the Markdown, validate syntax, or add further annotations.
   - The flow in `backend/execution.py` (if called by `main.py` or similar) would save the final `ResponseData` object as `response.json` in `outputs/archive/` and `outputs/latest/`.
   - The `api.py` endpoint then typically returns a simplified JSON response to the frontend, often containing just the `response_markdown` and `total_seconds`.
7. **Frontend Display:**
   - The React frontend receives the JSON response from the Backend API.
   - It extracts the Markdown content.
   - It uses a library like `react-markdown` to render the Markdown, displaying the LLM-generated test code and any accompanying text to the user in the web interface.

+ Add a custom footer

Pages 37

Find a page...

Show 22 more pages...

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/amosproj/amos2025ss04-ai-driven-testing.wiki.git
```