Design Document

IntelliTesting

Chengyan Song

Dec 2025

SETU

Final Year Project

1. Introduction

1.1 Purpose

This design document aims to detail the system architecture and design implementation of IntelliTesting. This project is an AI-driven VS Code extension dedicated to automating the generation and execution of unit tests, aiming to solve the problem of excessive time spent by independent developers and small teams on writing boilerplate test code.

2. System Architecture Overview

2.1 High-Level Architecture Design

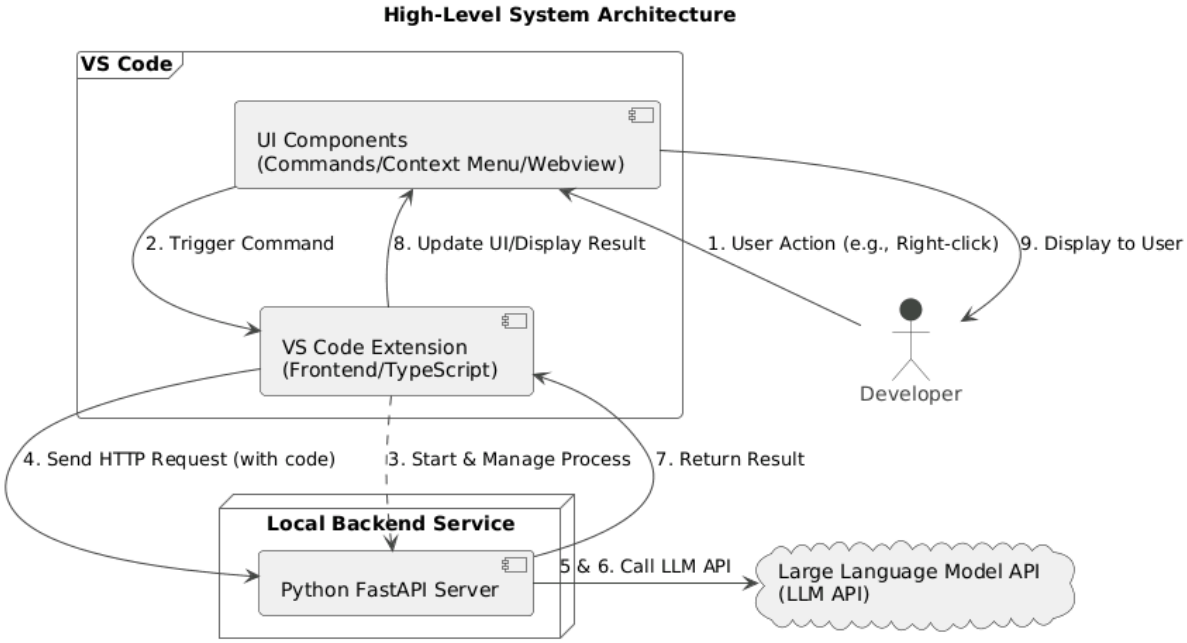

Figure 2.1: High-Level System Architecture (Decoupled Client-Server Model)

This diagram illustrates the high-level, component-based architecture of the AI-powered test generation tool. It showcases a decoupled client-server model designed to integrate seamlessly within the VS Code environment while separating the user interface from the heavy computational logic.

The system adopts a decoupled Client-Server model, separating the user interface from heavy computational logic.

- **Frontend (Client):** VS Code Extension. Developed in TypeScript, the primary

language recommended by Microsoft for VS Code extensions due to its strong typing and tooling support, though JavaScript is also supported **[1]**. It is responsible for UI interaction, context capture, and backend lifecycle management.

- **Backend (Server):** A Local Python FastAPI server running on the developer's machine (localhost). It is designed to run as a child process spawned by the extension, ensuring direct access to the local file system and verifying that sensitive source code logic remains local (excluding necessary LLM API calls).
- **AI Engine:** External Large Language Model (LLM) API (e.g., Google Gemini), accessed via the backend.

2.2 Why VS Code Extension Architecture?

- **Limitations of Web Apps:** According to research analysis, web applications cannot directly access the user's local file system or control the terminal environment, breaking the automation loop.
- **Benefits:** The VS Code extension architecture not only supports offline development (except for LLM calls) but also fully leverages existing IDE features like debugging and file navigation.

3. Functionality & Use Cases

3.1 Core Use Cases

The system is built around four core functional steps:

1. **F-01 Context-Aware Capture:** Automatically reads selected code.
2. **F-02 Intelligent Test Generation:** Uses AI to generate test code.
3. **F-03 Automated File Integration:** Writes test code to files.
4. **F-04 One-Click Execution & Validation (Run & Validate):** Automatically runs tests and reports results.

3.2 Core Workflow Phases

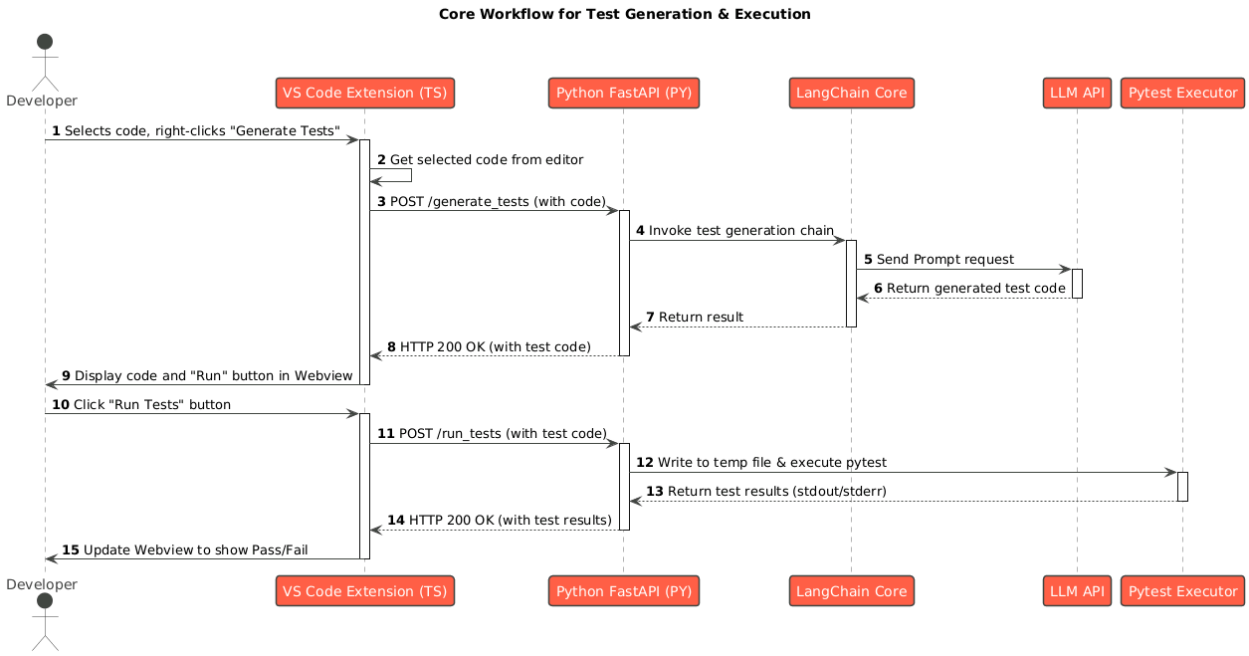

Figure 3.1: Core Workflow Sequence for Test Generation & Execution

This sequence diagram meticulously details the complete, step-by-step interaction flow for the primary functionality of the tool: generating unit tests and then executing them. It provides a chronological view of how control and data messages are passed between the different components, highlighting the two main phases of the operation.

The system handles user requests through two main phases: Generation Phase and Execution Phase.

- **Generation Phase:** User triggers command in IDE -> Extension sends HTTP request -> Backend calls LangChain/LLM -> Returns code -> Frontend displays it.
- **Execution Phase:** User clicks "Run" -> Backend creates temp file -> Invokes Pytest subprocess -> Captures Stdout/Stderr -> Frontend displays Pass/Fail status.

4. Backend Internal Architecture

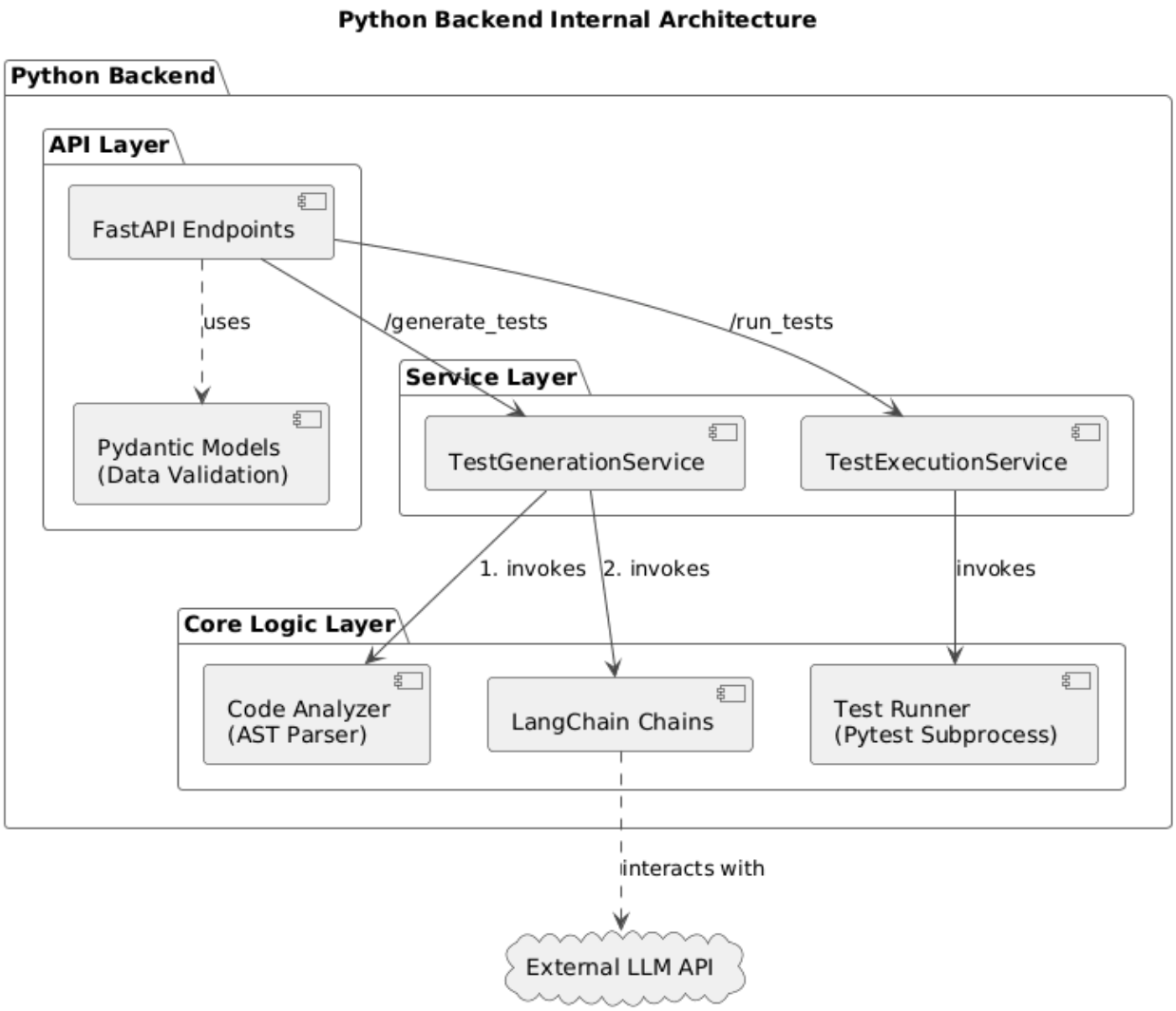

Figure 4.1: Python Backend Internal Architecture (Three-Layered Pattern)

This component diagram provides an in-depth look at the internal architecture of the Python backend service. It is designed using a classic three-layered architecture pattern, which is a software design best practice for creating robust, scalable, and maintainable applications by promoting a clear separation of concerns.

The backend adopts a three-layered architecture to ensure separation of concerns.

4.1 API Layer

- **Role:** The public-facing entry point responsible for handling incoming network requests and validating data.
- **Components:**

FastAPI Endpoints: Defines routes (e.g., /generate_tests).

Pydantic Models (Data Schemas): Defines strict data structures for API requests and responses. By enforcing these schemas, the system acts as a gatekeeper that automatically rejects malformed data (e.g., missing fields) at the entry point, preventing runtime errors in the Service Layer.

4.2 Service Layer

- **Role:** The central coordination layer containing business logic. It orchestrates workflows but delegates low-level tasks.
- **Components:**
  - **TestGenerationService:** Coordinates code analysis and LangChain invocation.
  - **TestExecutionService:** Manages the test execution process via the Test Runner.

4.3 Core Logic Layer

- **Role:** Contains specialized, reusable components that perform specific tasks.
- **Components:**

Code Analyzer (AST Parser): Instead of processing raw text strings, this component parses the source code into an Abstract Syntax Tree (AST).

- **Purpose:** To extract specific metadata (e.g., function signatures, class dependencies) for the LLM.
- **Why needed:** This ensures the AI receives structured context rather than noise, reducing token usage and improving generation accuracy compared to raw text analysis.

LangChain Chains (Dynamic Prompting): Manages the construction of dynamic prompts based on the context provided by the Code Analyzer.

- **Example of Use:** If the AST detects a complex Java class, the Chain selects a "Class-Level Testing" strategy and injects a JUnit 5 template. If a simple Python function is detected, it switches to a "Functional Testing" strategy with Pytest conventions.
- **Why needed:** This allows the system to adapt its strategy (Chain-of-Thought vs. Zero-Shot) based on code complexity, rather than using a static, "one-size-fits-all" prompt.

Test Runner (CLI Orchestrator): A secure wrapper component that invokes standard testing frameworks (e.g., Pytest, JUnit).

- **Role:** While xUnit executes the test logic, this runner manages the **subprocess lifecycle**.
- **Key Functions:** It isolates execution to prevent infinite loops from crashing the backend, captures `stdout/stderr` streams to display in the VS Code UI, and handles temporary file cleanup.
  -

5. Interface Design

5.1 Interaction Flow

IntelliTesting aims to provide a seamless UI experience:

- **Context Menu:** Developers can select "Generate Unit Tests" by right-clicking on the code.
- **Webview Panel:** Generation results are displayed via an embedded Webview, containing code previews and operation buttons.
- **Terminal Feedback:** Test execution results are fed back directly in the IDE's integrated terminal or Webview.

6. Technical Stack

Component	Technology	Role
Frontend	TypeScript, VS Code API	User Interface, Process Lifecycle Management
Backend Framework	Python, FastAPI	API Server, Business Logic Processing
AI Orchestration	LangChain	Prompt Management, Tool Chain Connection
AI Model	Gemini / LLM API	Code Generation and Inference
Execution Environment	Pytest / JUnit	CLI Test Runner
IPC	HTTP / REST	Communication between Extension and Local

		Server

References

[1] Microsoft. (2025). *Your First Extension - Visual Studio Code API*. Available at: https://code.visualstudio.com/api/get-started/your-first-extension