# Design Document

## IntelliTesting: An Agentic Framework for Specification-Driven Unit Test Generation

*Chengyan Song*

*Feb 2026*

*SETU*

*Final Year Project*

# 1. Introduction

## 1.1 Background

Software testing is a critical phase in the software development lifecycle (SDLC), consuming up to 50% of development time. While automated testing frameworks (JUnit, PyTest) exist, the creation of test cases remains a manual, cognitive-heavy task. The advent of Large Language Models (LLMs) offers a potential solution for automated code generation.

## 1.2 Problem Statement

Despite their capabilities, naive applications of LLMs for testing suffer from critical defects:

1. Ambiguity & Assumption: When business logic is implicit, LLMs "guess" the intended behavior based on the implementation code. This leads to tautological testing, where tests merely confirm that the code does what it does, not what it should do.
2. Hallucination: LLMs frequently generate code that references non-existent files, incorrect package paths, or phantom libraries, rendering the code uncompilable.
3. Lack of Verification: Generated code is often treated as "final," skipping the crucial step of runtime verification in the user's local environment.

## 1.3 Project Objectives

This project, IntelliTesting, aims to develop a VS Code extension that evolves beyond simple code completion into a Quality Assurance Agent.

- To implement an Interactive Verification workflow: Shifting from "Auto-Generate" to "Verify-then-Generate," ensuring the developer clarifies edge cases before code creation.
- To enforce Specification-Driven testing: Prioritizing natural language requirements (Oracles) over source implementation to detect logical regression bugs.
- To ensure Execution Safety: Running tests in a local, ephemeral sandbox to verify correctness without polluting the project workspace.

## 2. Technology Stack

### 2.1 Backend (The Brain)

- Language: Python 3.12+
- Framework: FastAPI (High-performance async REST API)
- AI Orchestration: LangGraph (State Machine based Agent orchestration) & LangChain.
- LLM: Google Gemini 2.5 Flash (via Google GenAI SDK).
- Engineering Pattern: Factory Pattern (Language Strategies) & Command Pattern.

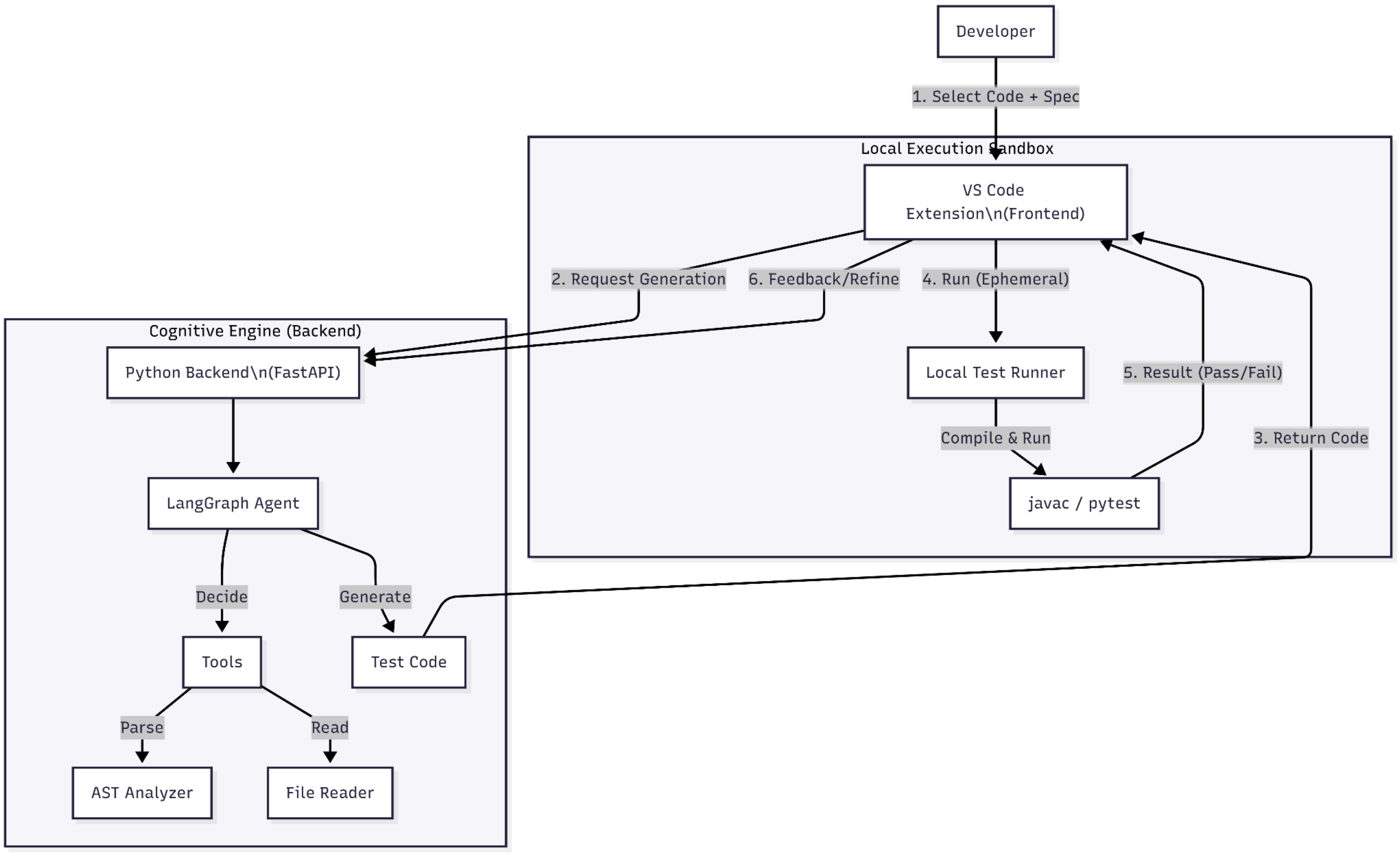
### 2.2 Frontend (The Body)

- Platform: Visual Studio Code Extension API.
- Language: TypeScript.
- UI Framework: Custom HTML/CSS/JS injected into VS Code Webview (React-style component architecture without heavy bundlers).
- Communication: JSON-RPC over HTTP (fetch) + Message Passing (postMessage).

## 3. System Architecture

IntelliTesting adopts a Client-Server Architecture where the VS Code Extension acts as a thin client for UI and File I/O, while the Python Backend handles complex reasoning and state management.

### 3.1 Architecture Diagram



## 4. Backend Design: The Cognitive Engine

The backend is the "brain" of IntelliTesting, responsible for analyzing code, reasoning about testing strategies, and orchestrating the test generation process. It is built on Python and leverages the LangGraph library to implement a stateful, cyclical agentic workflow.

### 4.1 LangGraph State Machine Architecture

Unlike linear "Chain" architectures (A -> B -> C), IntelliTesting uses a State Machine (Graph) architecture. This allows the agent to loop, retry, and make dynamic decisions based on tool outputs.

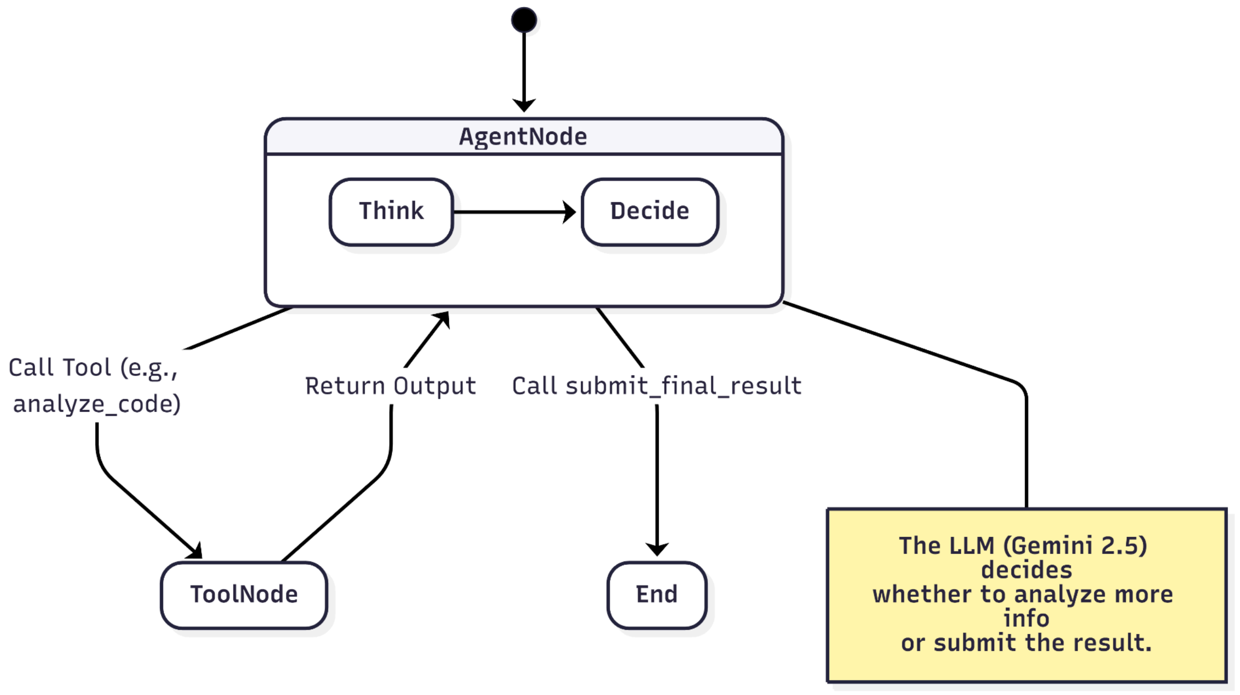
### 4.1.1 State Definition

The agent's memory is maintained in a typed state object, ensuring type safety and structured data flow across nodes.

```
class AgentState(TypedDict):
    messages: Annotated[List[BaseMessage], operator.add]  # Append-only message history
    file_content: str         # The full content of the source file
    selected_code: str        # The specific method/class selected by the user
    language: str             # 'java' or 'python'
    interactive_questions: Optional[List[str]] # Questions generated for the user
    final_test_code: str      # The generated test code
```

### 4.1.2 The Agent Loop (Workflow Diagram)

The following Mermaid diagram illustrates the cyclical execution flow of the agent:


AgentNode
Think
Decide
Call Tool (e.g., analyze_code)
Return Output
Call submit_final_result
ToolNode
End
The LLM (Gemini 2.5) decides whether to analyze more info or submit the result.

1. Start: The workflow is initialized with the initial_prompt (containing code and spec) injected into messages.
2. AgentNode: The LLM processes the current state. It can choose to:
   - Call analyze_source_code: If it needs to understand dependencies (e.g., imports).

    - Call submit_final_result: If it has generated the test code or questions.
3. ToolNode: Executes the requested tool function in Python and appends the result (ToolMessage) back to the state.
4. Loop: The graph returns to AgentNode with the new tool output, allowing the LLM to reason based on new information.

## 4.2 Design Patterns in Backend

To ensure extensibility and clean code, the backend heavily utilizes established Software Engineering design patterns.

### 4.2.1 Factory Pattern (LanguageFactory)

The system supports multiple languages (Java, Python) without coupling the core logic to specific implementations.

- Role: The LanguageFactory class acts as a central dispatcher.
- Mechanism:

```
class LanguageFactory:
    @staticmethod
    def get_strategy(language: str) -> LanguageStrategy:
        if language == "java":
            return JavaStrategy()
        elif language == "python":
            return PythonStrategy()
        else:
            raise ValueError("Unsupported language")
```

- Benefit: Adding support for TypeScript or Go only requires creating a new Strategy class and updating the Factory, adhering to the Open/Closed Principle.

### 4.2.2 Strategy Pattern (`LanguageStrategy`)

Each language requires unique parsing logic (identifying packages) and prompt templates (JUnit vs PyTest syntax).

- Interface: LanguageStrategy defines the contract (analyze_code, get_test_prompt_template).

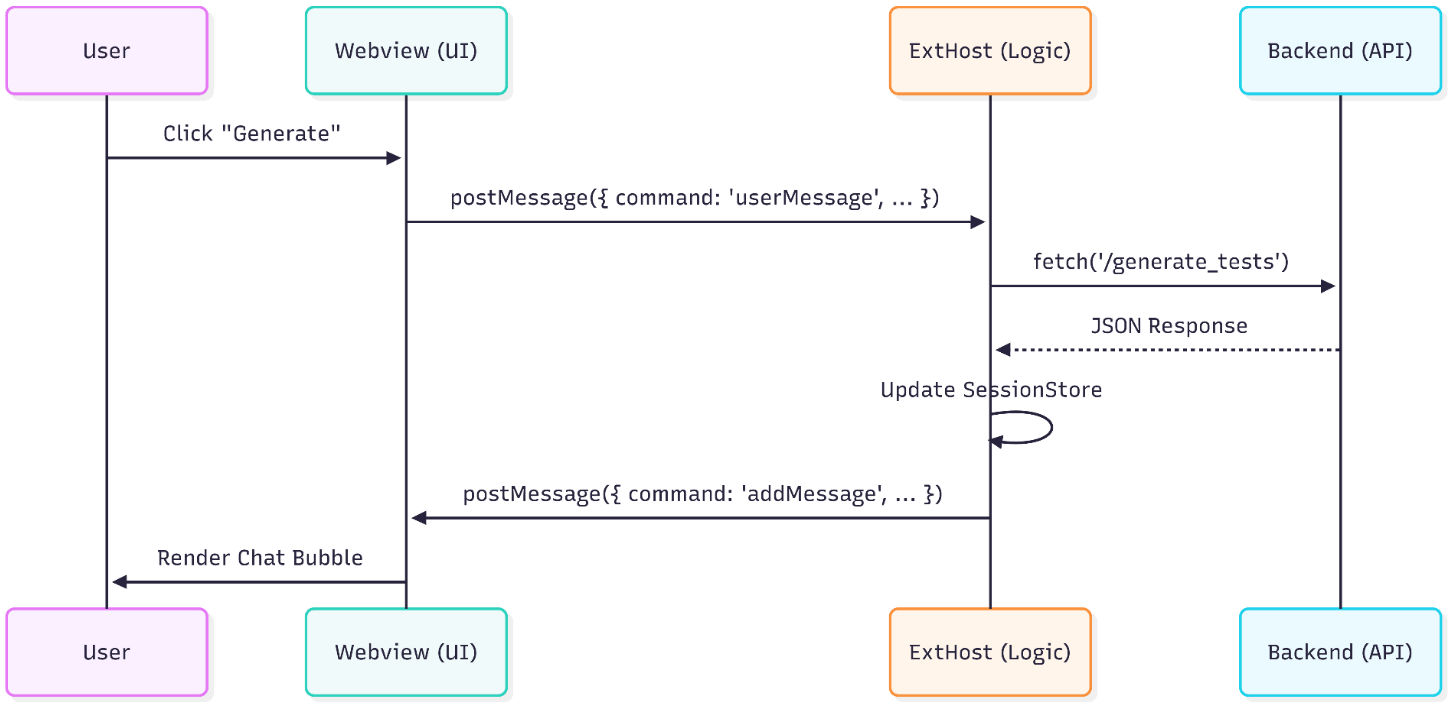
- Concrete Strategy (JavaStrategy): Implements logic specific to Java, such as parsing package declarations and mapping src/main to src/test.
- Usage: The TestGenerationService depends only on the abstract LanguageStrategy, making the generation logic language-agnostic.

## 5. Frontend Design: Interaction, State & Execution

The frontend is a sophisticated VS Code Extension built with TypeScript. It acts not just as a view layer, but as a robust execution environment and state manager.

### 5.1 Architecture: The Event Bus Model

Communication between the VS Code Extension Host (Node.js) and the Webview (Browser Sandbox) is handled via an asynchronous Event Bus.



### 5.2 Design Patterns in Frontend

Just like the backend, the frontend employs architectural patterns to manage complexity.

#### 5.2.1 Runner Factory Pattern (RunnerFactory)

To execute tests locally, the frontend must invoke different command-line tools (javac/mvn for Java, pytest for Python).

- Implementation: src/runners/RunnerFactory.ts
- Logic:

```
export class RunnerFactory {
    static getRunner(language: string): ITestRunner {
        switch (language) {
            case 'java': return new JavaRunner();
            case 'python': return new PythonRunner();
            default: throw new Error("Unsupported");
        }
    }
}
```

- Benefit: The UI component (extension.ts) doesn't need to know how to run tests. It simply asks the factory for a runner and calls .run().

#### 5.2.2 Strategy Pattern (Runners)

- JavaRunner Strategy: Handles complex Java compilation steps: creating a temp directory, constructing the Classpath (finding JUnit jars), compiling source + test files, and parsing JUnit XML output.
- PythonRunner Strategy: Handles creating a virtual environment (optional) or running pytest directly and parsing stdout.

## 5.3 Advanced Session State Management

A critical requirement was preserving chat history and context when the user switches between different files (tabs) in the IDE.

Data Structure: The sessionStore is a global Map that acts as an in-memory database for the extension session.

```
// Map Key: File Path (e.g., "src/main/java/App.java")
const sessionStore = new Map<string, {
    history: ChatMessage[],  // Array of user/AI messages
    spec: string             // The active specification for THIS file
}>();
```

Lifecycle Logic:

1. Switch Tab: When the user switches tabs, the Webview is destroyed by VS Code.
2. Re-Activate: When the user clicks the extension icon again, createOrShowWebview is called.
3. Restore: The webviewReady listener triggers. It queries sessionStore using the current file path.
4. Replay: If data exists, it sends a sequence of addMessage events to the Webview, instantly reconstructing the conversation and Spec context.

This design ensures Context Isolation: Specifications for OrderService never bleed into UserService, solving a common hallucination problem in AI assistants.

### 5.4 UI Implementation Details

The UI is built using vanilla HTML/CSS/JS to keep the extension lightweight, but it mimics a modern React application structure.

- Dynamic Rendering: Messages are appended to the DOM dynamically using document.createElement.
- Security: User input is sanitized via a parseMarkdown function that escapes HTML tags before rendering custom badges (like "📎 SPECIFICATION ATTACHED"), preventing XSS attacks.
- Live Feedback: The "Thinking" indicator uses a setInterval loop to cycle through text ("Scanning...", "Drafting..."), providing immediate visual feedback to the user.

## 6. Key Feature Implementation

### 6.1 Interactive Verification Mode

This feature addresses the "Black Box" problem.

1. Trigger: User selects code without providing a Specification.
2. Backend Logic: The Agent detects spec_context = "NO SPECIFICATION".
3. Agent Action: Instead of generating code, the Agent calls submit_final_result with test_code=None and a list of interactive_questions.
4. Frontend Render: The UI displays a warning card: "⚠️ Interactive Check Required" with the list of questions.
5. User Response: The user answers the questions in the chat.
6. Loop: The Backend combines the original code + questions + user answers to form a "Derived Specification" and generates the code.

### 6.2 Specification-Driven Oracle

This allows the tool to find bugs.

- Mechanism: The System Prompt explicitly instructs the model: "If the Source Code conflicts with the Specification, the Specification is the Oracle (Truth). Generate a test that expects the Specification's outcome."
- Result: If code has if (x > 10) but Spec says "x >= 10", the test assert(10) == true will be generated. The test fails (Red), alerting the developer to the bug.

### 6.3 Ephemeral Sandbox Execution

To ensure "Zero Pollution":

1. Temp Dir: fs.mkdtempSync creates a directory in the OS temp path.
2. Mirroring: The runner replicates the necessary folder structure (package paths).
3. Classpath Construction: It scans the user's workspace for .jar libs (JUnit) and bin/ or target/ folders.
4. Isolation: The test runs in a child process.
5. Destruction: The temp directory is recursively deleted immediately after execution, regardless of success or failure.

## 7. Implementation Challenges & Solutions

### 7.1 The "Ghost Annotation" Hallucination

Issue: The LLM persisted in adding @intellitesting... annotations that looked like file paths, likely leaking from internal context traces. Solution: A multi-layered cleaning pipeline.

1. Prompt Negative Constraint: "DO NOT use file paths as annotations."
2. Structured Output: Moving code to a JSON field reduced text-completion noise.
3. Regex Firewall: A final post-processing step in Python:

```
test_code = re.sub(r'^.*@.*[\\/].*$\n?', '', test_code,
flags=re.MULTILINE)
```

This aggressively removes any line containing an @ symbol followed by a path separator, guaranteeing clean output.

### 7.2 Webview Re-generation Loop

Issue: Switching tabs caused the Webview to reload, sending webviewReady, which wrongly triggered a new backend generation call. Solution: Logic separation.

- webviewReady now only replays history from sessionStore.
- Backend generation is only triggered by the explicit "Generate" command or user message event.

## 8. Critical Analysis & Limitations

### 8.1 Strengths

- Architecture: Decoupled, event-driven, and scalable.
- Safety: Local sandbox ensures no side effects on the developer's machine.
- Accuracy: Spec-driven approach significantly reduces "lazy" tests that just mimic buggy code.

### 8.2 Limitations

- Integration Testing: The current sandbox creates a fresh environment. It cannot easily test code that relies on a running local database or complex Spring Boot Dependency Injection without significant configuration.
- Context Window: For extremely large files (>1000 lines), the context window may truncate relevant logic, although Gemini's large window mitigates this.

## 9. Conclusion

IntelliTesting successfully evolves from a simple code generator to a Quality Assurance Agent. By enforcing structured outputs, implementing strict state management, and introducing the Interactive Verification workflow, it addresses the fundamental "trust issue" in AI-generated code. It not only writes tests but actively helps developers refine their requirements, embodying the principles of modern software engineering.