Functional Specification

IntelliTesting

Chengyan Song

Dec 2025

SETU

Final Year Project

1.Introduction

This specification provides all functionalities and requirements for IntelliTesting. It offers detailed descriptions and illustrations to aid in directing the creation and validation of IntelliTesting.

IntelliTesting seeks to fulfil the growing need for enhanced developer productivity and code quality by employing innovative AI-driven automation techniques. The application focuses on fixing common issues with boilerplate unit test writing and providing a workflow that enhances the development experience by integrating test generation, execution, and validation directly into the IDE. IntelliTesting ensures that developers not only write application code but also generate and verify the corresponding tests with minimal interruption.

IntelliTesting aims to create an engaging and effective software development experience that empowers users to improve their code quality confidently and competently, achieving a Zero-Interruption Workflow[1].

2.Project Description

2.1 Project Objective

The goal of this project is to implement all the requirements and features mentioned in this document. Deliver a good application to create a good user experience

2.2 Target Audience

The primary audience for this project is independent software developers or small development teams. They often spend a lot of time testing their code, and IntelliTesting can effectively solve this problem, improving the development efficiency of independent developers or teams.

2.3 Problem Statement

Independent developers and small teams operate with extremely limited time and resources. The current process of manual unit testing forces these lean operations to allocate critical development hours—which should be spent on features or growth—to the repetitive, non-differentiating task of writing boilerplate test code. This overhead creates significant development friction, slowing down iteration cycles and increasing the risk of technical debt. The core problem is that this manual burden directly prevents small teams from achieving the high velocity and full test coverage necessary to compete effectively, forcing a difficult and constant trade-off between speed and code quality.

3. Functionality

3.1 Core Features

1. **Real-time Test Case Generation:**

   Users can generate real-time automated test cases by just clicking a few buttons without having to write additional test code.

2. **Automatic Test Execution:**

   After the test cases are automatically generated, the test is automatically run through the terminal to improve the test efficiency

3.2 Additional Features

1. **Context-Aware Capture:**

   Automatically read all code files and configuration files in the project to better analyze the AI model and generate more appropriate test code

2. **Custom Configuration:**

   Providing developers with granular control over backend endpoints, test execution commands, and project test paths to ensure enterprise readiness and flexibility.

3.3 Dependencies

Host Environment: The operating system and integrated development environment (IDE) version required to run the VS Code extension and underlying test execution.

Frontend Runtime Environment: The scripting runtime required to run VS Code extension code, handling user interface and file operations.

AI Service Platform: The cloud service or on-premises backend infrastructure that hosts and runs Large Language Models (LLMs), providing core inference capabilities.

API Gateway & Security: The HTTPS/TLS infrastructure and authentication gateway for secure communication between the frontend and AI services.

Project Execution Environment: The target test executor and underlying command-line interface (CLI) installed in the developer's project, used to run and validate generated tests.

3.4 Usability

Context-Aware Automation: Ensure the system automatically determines testing intent (e.g., capturing the selected code), requiring minimal manual input from the user.

Instant Visual Feedback: Immediately open the newly created test file and clearly display execution results in a focused terminal within the IDE for instant verification.

Non-Disruptive Integration: Use robust VS Code APIs[2] for file writing and terminal management, ensuring the tool's actions do not interfere with the developer's ongoing work or undo history.

Simple Configuration: Manage all required settings (like backend URL and run command) centrally within the standard VS Code settings interface, eliminating external configuration files.

3.5 Reliability

Non-Crashing Integration: The VS Code plugin must isolate itself from backend failures (like network timeouts or 500 errors), ensuring it never crashes or freezes the developer's IDE environment.

Guaranteed File Integrity: The system must ensure test file creation (F-03) is clean and safe, preventing any risk of leaving corrupted or partially written files in the user's project, thereby maintaining project stability.

3.6 Performance

Acceptable Test Generation Time: The plugin must respond with the generated test code (F-02) within a reasonable time, targeting under 2minutes from command trigger to code receipt, to minimize workflow interruption.

Low IDE Overhead: The plugin must ensure its background processes and code analysis (F-01) have minimal impact on VS Code's CPU and memory usage when idle, keeping the developer's environment responsive.

3.7 Security

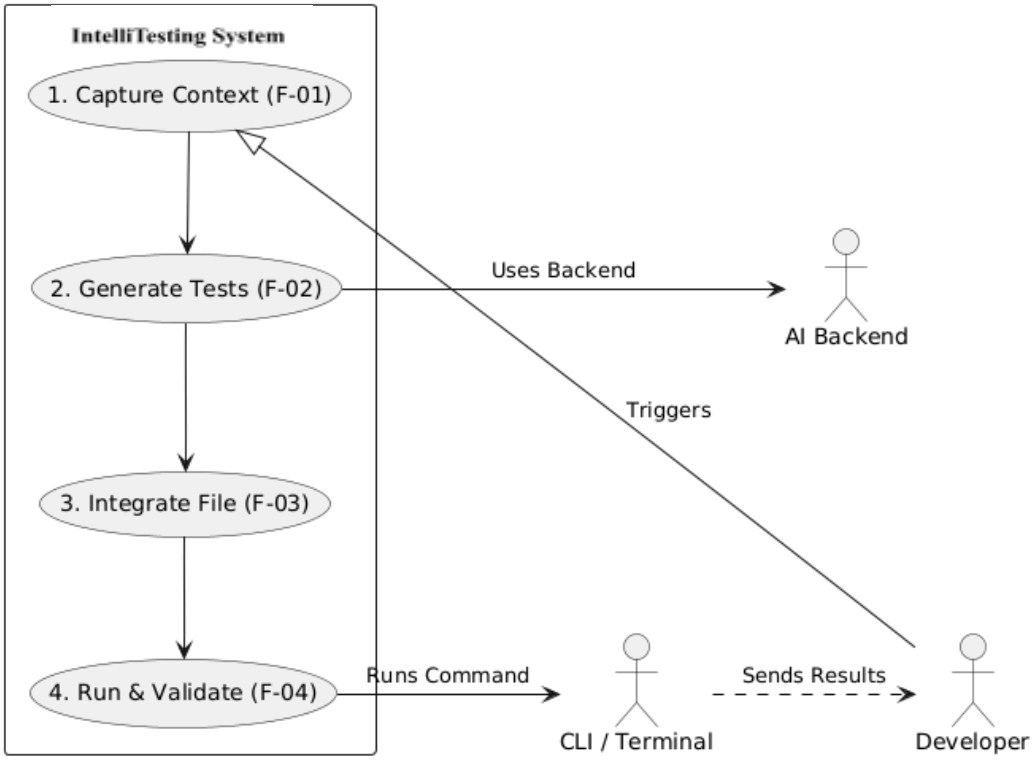
Ensure user data is private and protected: Comply with GDPR[3] to keep users' data safe

3.8 Technology Stack

Category	Key Technologies
Frontend/IDE Extension	TypeScript (TS), VS Code Extension API, Webview
Backend/Server	Python, FastAPI[4]
AI Module	LLM API, LangChain[5] Core
Execution Environment	Pytest or Junit Executor, Terminal/CLI
Communication	HTTP Protocol

4. Project Architecture

4.1 Use Case Diagram



4.2 Brief Use Case

Use Case ID	Use Case Name	Brief Description
F-01	Context-Aware Capture	The developer triggers the command; the plugin captures the selected code, language, and configuration for backend delivery.
F-02	Intelligent Test Generation	The AI Backend receives the code context, using the LLM to generate test code strictly compliant with the specified framework.
F-03	Automated File Integration	The plugin parses the AI response, atomically creating and writing the generated test file to the correct project path.
F-04	One-Click Execution & Validation	The plugin constructs and executes the targeted test command in the CLI to run the new file and retrieve instant pass/fail feedback.
UX-01	Configuration Management	The Developer uses VS Code settings to configure the AI backend URL, default test framework, and project file paths.

5. Usage Scenarios & Testing Strategies

To accommodate different development contexts, IntelliTesting supports two distinct modes of operation based on the availability of functional requirements. This distinction determines whether the system acts as a simple code generator or a requirement validator.

5.1 Scenario A: No Specification (Code-Driven Generation)

Input: Source Code only.

System Logic: In the absence of external specifications, the AI treats the current implementation as the "Source of Truth." It analyzes the existing logic paths and generates test cases that ensure the code executes without runtime errors.

Role: Regression Guard.

Outcome: The generated tests will typically PASS against the current code. The primary value here is to create a safety net for future refactoring, ensuring that code changes do not break existing (assumed correct) behavior.

5.2 Scenario B: With Specification (Requirement-Driven Validation)

Input: Source Code + Specification Context (e.g., Javadoc, docstrings, or requirement comments).

System Logic: The AI treats the Specification as the "Source of Truth." It generates test cases specifically designed to assert the conditions defined in the requirements, ignoring the actual implementation details during generation.

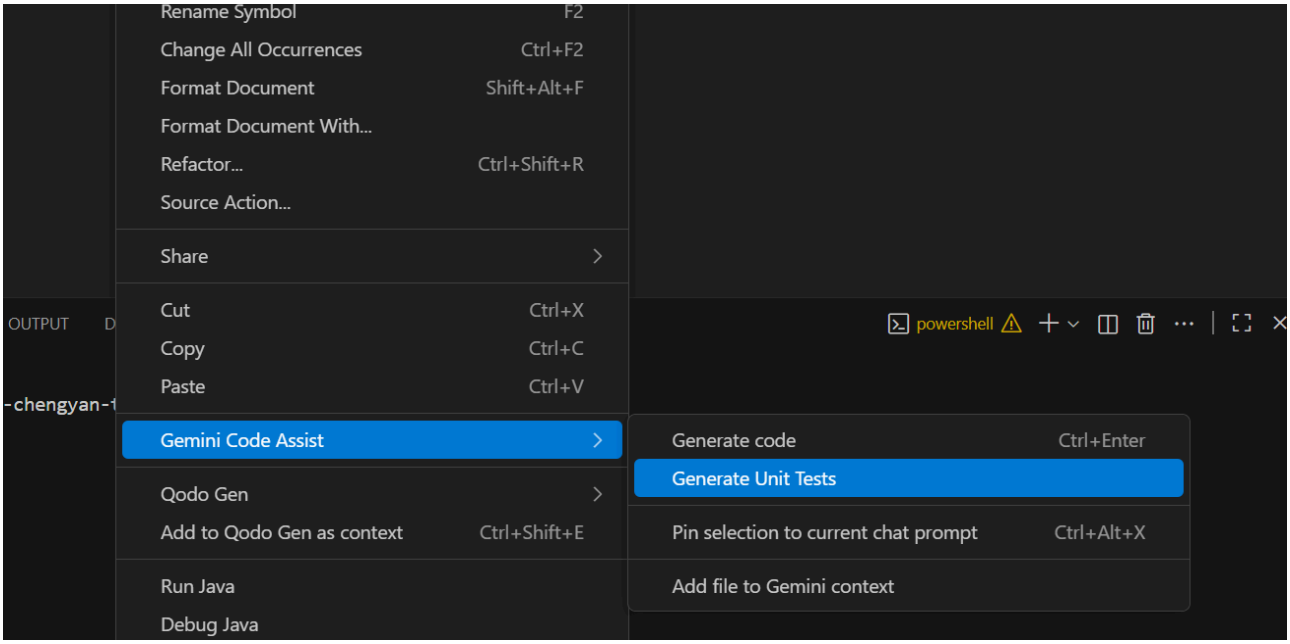
Role: Correctness Validator.

Outcome: The generated tests serve as a judgment mechanism. If the implementation contradicts the specification (e.g., a function returns null when the spec says it should throw an Exception), the generated tests will FAIL. This mode actively detects "Intent-Implementation Mismatches," fulfilling the critical role of automated quality assurance.

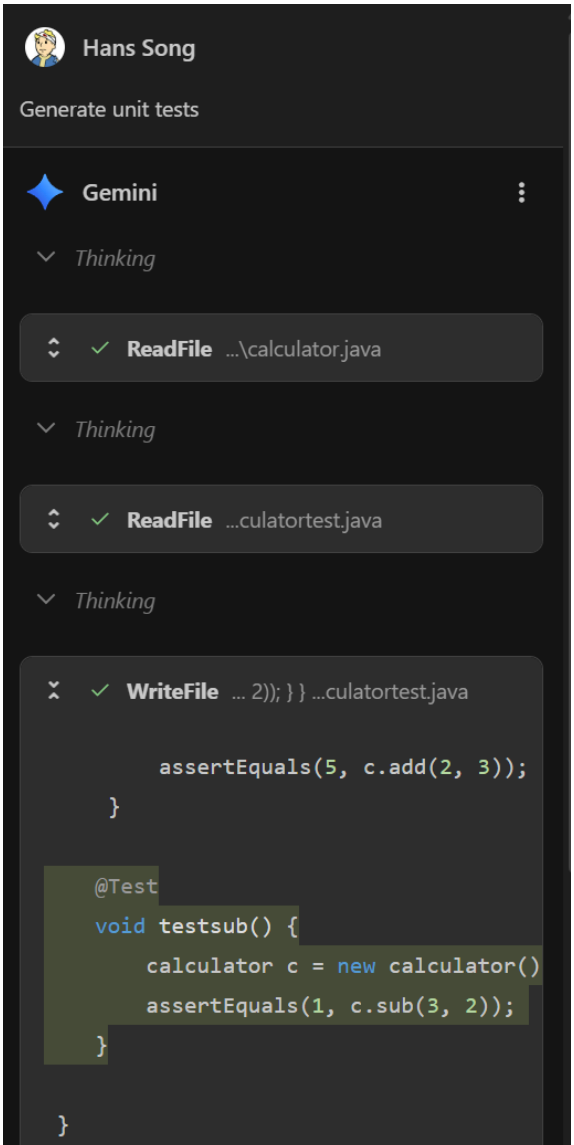
6.Interface Demonstration

6.1 Test Case Generation

```java

public class Calculator{

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return a / b;
    }


}

```



1. When the user right-clicks, a menu will pop up, which contains the section to which the plug-in belongs. Placing the mouse over it will enter the next level and display the generated test cases.



2. Once the user clicks on Generate Test, a chat box will appear on the right side of the interface. The execution, status, and progress of all operations will be displayed in this chat box. At the same time, the plug-in will read the code context and pass it to the backend AI model for thinking and generation. Once the generation is complete, a prompt will appear in the chat box on the right, indicating that the generation is complete and whether to apply it. Click Apply, and the plug-in will create a test file or modify and optimize the existing test file.

3. When the generation operation is completed, a test button will pop up in the chat box. Click the button and the plug-in will create a new terminal to run the test framework required by the code, such as Junit, and receive the return information and results to the user.

7.Reference

[1]**Zero-Interruption Workflow:** A development philosophy and design goal to eliminate or minimize contextual shifts and manual tasks by automating repetitive processes (like test generation and execution), allowing the developer to remain in a state of flow within the IDE.

[2]**VS Code APIs:**

vscode.WorkspaceEdit (or applyEdit): API used for atomic file operations (create, insert, delete). Ensures file changes are added to the Undo/Redo history, preventing file corruption and preserving the developer's work integrity.

vscode.window.createTerminal() and terminal.sendText(): APIs used to create a dedicated, isolated terminal (e.g., "IntelliTesting Runner") and securely send the execution command. This guarantees that test execution does not disrupt the developer's main working terminals.

https://code.visualstudio.com/api/references/vscode-api

[3]GDPR. General Data Protection Regulation. https://eur-lex.europa.eu/eli/reg/2016/679/oj

[4]FastAPI. https://fastapi.tiangolo.com/

[5]LangChain. https://python.langchain.com/docs/introduction/