IntelliTesting

An AI-Driven Test Case Generator

Final Year Project — Project Report

Chengyan Song
South East Technological University (SETU)
Supervisor: Christophe Meudec

December 2025

1. Introduction **3**
2. Project Achievements **4**
2.1 Addressing the Oracle Problem 4
2.2 Interactive Verification Before Code Generation 4
2.3 Self-Verifying Agent Loop 4
2.4 User Feedback and Iteration 5
3. Unmet Objectives **6**
3.1 VS Code Marketplace Publication 6
3.2 Automated Dependency Resolution 6
3.3 Test Session Persistence 6
4. Challenges and Resolutions **7**
4.1 LLM Hallucination of File Path Annotations 7
4.2 Webview State Loss on Editor Tab Switch 7
4.3 JUnit Classpath Construction Without a Build System 7
4.4 Agent Request Timeout on Slow Connections 8
4.5 Cross-File Context Contamination 8
5. Testing and Validation **9**
5.1 Functional Metrics 9
5.2 AI Branch Coverage Evaluation 9
5.3 Hallucination Rate Before and After Mitigation 10
6. Reflections and Lessons Learned **11**
6.1 Verification Is the Core Problem, Not Generation 11
6.2 Platform-Specific Constraints Cannot Be Inferred from Documentation 11
6.3 Research Must Be Validated with Prototypes 11
6.4 User Testing Surfaces Problems That Self-Testing Cannot 11
6.5 Integration Testing Should Not Be Deferred 12
7. Future Recommendations **13**
7.1 Button State Management and UI Polish 13
7.2 Streaming Architecture for Agent Progress 13
7.3 Dependency Resolution via Language Server Protocol 13
7.4 Session Persistence via globalState 13
7.5 Marketplace Publication with Mandatory API Key Setup 13
8. Conclusions **14**

1. Introduction

This report reflects on the complete development of IntelliTesting, a VS Code extension that automates unit test generation, execution, and validation using an AI agent backend. It examines what was delivered against the original specification, the technical and design challenges encountered during implementation, and the lessons drawn from the experience. The Functional Specification, Design Document, and Research Document contain the detailed technical descriptions and architectural diagrams that this report does not repeat.

The project was motivated by a concrete problem: independent developers and small teams spend a disproportionate share of their time writing boilerplate test code. The hypothesis was that a large language model, integrated directly into the IDE, could automate this work without interrupting the developer's workflow. What the project revealed is that this hypothesis is true in a narrow sense — the model can generate syntactically valid test code — but false in a more important sense: without systematic verification and grounding, the output is unreliable enough to be counterproductive. Most of the engineering effort in this project went not into generating tests, but into making the generated tests trustworthy.

2. Project Achievements

The project delivered a fully functional VS Code extension with a cloud-deployed AI backend. All four mandatory use cases defined in the Functional Specification were implemented and demonstrated. This section evaluates the most significant achievements not in terms of feature completion, but in terms of the problems they solve and how they compare to the original intentions.

2.1 Addressing the Oracle Problem

The oracle problem — the difficulty of knowing whether generated tests assert the right behaviour rather than merely the current behaviour — was identified in the Research Document as the central limitation of AI-driven test generation. A naive AI test generator will produce tests that pass even on buggy code, because the model infers expected outputs from the implementation rather than from any specification of correct behaviour. This creates a false sense of security.

IntelliTesting's specification-driven mode directly addresses this. When the user provides a natural language requirements document, the backend uses it as the ground truth and generates tests that assert the specified behaviour. If the implementation contradicts the specification, the generated tests are deliberately written to fail, surfacing the defect. This is a meaningful inversion of the standard AI testing workflow and was the most technically and conceptually ambitious feature of the project.

The limitation is that the feature is only as useful as the specification the user provides. For greenfield code with clear requirements, it works well. For legacy code — which has no accompanying specification and is often the most in need of test coverage — the system falls back to the interactive verification mode, which is less powerful but still more rigorous than blind generation.

2.2 Interactive Verification Before Code Generation

Rather than generating test code immediately, the system first produces a structured test plan — a set of human-readable scenarios, each describing an intent, input conditions, and expected output — and presents it for the developer's review. Only the scenarios the developer approves are passed to the code generation step.

This addresses a subtler form of the oracle problem: even when no specification is available, the developer often has implicit knowledge about what the correct behaviour should be. By surfacing the model's assumptions as explicit, reviewable cards before committing them to code, the system gives the developer a natural point at which to correct misunderstandings. In user testing, this was the feature most positively received, with one tester noting that seeing the test plan before the code made them significantly more willing to trust the generated output.

2.3 Self-Verifying Agent Loop

The most impactful engineering decision made during the project was the implementation of a self-verifying agentic loop. After generating test code, the backend agent automatically executes the tests in a sandboxed environment and, if they fail due to compilation errors or runtime issues, attempts to diagnose and correct the problem — up to a maximum of eight iterations. Only when the tests pass locally does the agent submit them as the final result.

This was not part of the original specification. It was introduced after observing that approximately 40% of generated test files contained hallucinated imports, incorrect package paths, or invalid method

signatures that caused immediate compilation failure. The self-verification loop reduced this failure rate to under 5% in subsequent testing. The decision to build verification into the generation loop, rather than treating it as a post-processing step, proved to be architecturally significant: it transformed the system from a code suggestion tool into a working test generator.

2.4 User Feedback and Iteration

The extension was shared with a small group of peers for informal user testing. Two pieces of feedback led to concrete changes. First, the extension lacked a toolbar icon, making it difficult to discover within VS Code's interface; an icon was subsequently added to the activity bar. Second, testers noted that the "Generate Tests" button remained active even when no code had been selected, leading to confusing empty-context generations. This was identified as a usability issue that should be resolved in a future release by disabling the button when the selection is empty.

The fact that both pieces of feedback were interface-level rather than functional reflects positively on the underlying reliability of the system, but also highlights the importance of user testing for surfacing usability issues that are invisible to the developer.

3. Unmet Objectives

Three requirements from the original specification were not fully delivered within the project timeline. Each is described below with an account of why it was not completed.

3.1 VS Code Marketplace Publication

The extension was fully packaged as a .vsix installer for private distribution but was not published to the public VS Code Marketplace. The technical packaging was completed; the obstacle was operational rather than technical. A publicly listed extension could be installed by an unlimited number of users, and the current implementation shares a single Gemini API key across all requests. During informal testing, the shared daily quota was exhausted within two days by a group of fewer than ten testers. Without a per-user API key enforcement at the marketplace level, publication would have rendered the service unusable within hours of listing.

The settings layer already supports user-provided API keys as a bypass mechanism, so the groundwork for a safe publication is in place. The remaining step — making an API key mandatory at installation through a VS Code walkthrough — was not completed within the project timeline.

3.2 Automated Dependency Resolution

The Functional Specification described a context-aware capture step (F-01) that would automatically identify and inject files imported by the target class into the AI prompt. In the delivered implementation, the agent can read specific files using a tool call, but the decision of which files to read is left to the model rather than determined by static analysis of the import graph.

The practical consequence was identified during testing with real-world Java projects: classes that depend on custom interfaces or abstract base classes produced test code with incorrect method signatures, because the model was inferring the interface contract from the class name alone rather than reading the actual interface definition. A correct implementation would require parsing the Java or Python import declarations and resolving them against the project's source tree before constructing the AI prompt. This was scoped out after the estimated implementation time exceeded what the project timeline permitted.

3.3 Test Session Persistence

Conversation history and attached specification documents are maintained in memory for the duration of the VS Code session but are not persisted to disk. Closing and reopening VS Code clears all context. This was an oversight in the original design: the per-file session store was added to solve the Webview re-generation bug (discussed in Section 4) and was not extended to include persistence. VS Code's globalState API would support this with minimal additional code and represents a straightforward improvement for a future release.

4. Challenges and Resolutions

This section documents the significant technical problems encountered during implementation, the process of diagnosing them, and the solutions adopted. These are problems that were not anticipated in the design phase and required unplanned development effort.

4.1 LLM Hallucination of File Path Annotations

The most persistent bug in the project was the model's consistent tendency to insert file path comments at the top of generated test code. A typical example would be a line such as:

```
// File: src/test/java/com/example/NonExistentTest.java
```

These annotations referenced paths that did not exist in the user's project. The frontend used this path to determine where to write the test file; when the path was invalid, the file was either written to the wrong location or not written at all. The problem appeared consistently regardless of how the system prompt described the output format, and attempts to eliminate it through instruction alone were ineffective.

The root cause is that Gemini was trained on a large corpus of code files that typically include file path headers, and the model reproduces this pattern reflexively. The resolution required three concurrent changes: a regular expression post-processor was added to the backend to strip any line matching the pattern of a file path comment before the response reaches the frontend; the system prompt was extended with explicit negative examples showing the prohibited pattern; and the suggested file path was extracted from a dedicated structured JSON field, removing the frontend's dependency on any path embedded in the code itself. The combination proved more reliable than any single measure.

4.2 Webview State Loss on Editor Tab Switch

VS Code destroys the DOM of a Webview panel when it loses focus and reconstructs it when focus returns. This is documented behaviour, designed to reduce memory usage when panels are not visible. Its consequence for IntelliTesting was that every time the user switched to a different editor tab and back, the entire chat history was erased. This was discovered during self-testing when a multi-turn conversation was lost after switching tabs to check a source file.

The solution was to relocate all session state from the Webview (browser sandbox) to the Extension Host (the persistent Node.js process). The Webview became purely a rendering layer with no authoritative state of its own. When a Webview is recreated after focus is restored, the Extension Host immediately replays the full session into it via postMessage, restoring the conversation transparently. A secondary benefit of this change was that it made per-file session isolation straightforward to implement: the Extension Host maintains a Map keyed by file URI, so switching between open files automatically restores the correct session context for each file.

4.3 JUnit Classpath Construction Without a Build System

Executing JUnit tests requires junit.jar and hamcrest-core.jar to be on the Java classpath. These libraries are not part of the standard Java Development Kit and are not guaranteed to be present on the developer's machine outside a Maven or Gradle project. The initial implementation assumed they would be available globally and produced ClassNotFoundException errors on clean machines.

The solution was to bundle the required JARs with the extension package itself and construct the classpath dynamically at runtime by combining the bundled JARs with any .jar files found in the project's

lib/ directory. The decision to bundle rather than download was deliberate: downloading dependencies at runtime introduces a network dependency and a version management problem. Bundling ensures reproducible behaviour regardless of the developer's environment. The trade-off is a slightly larger extension package size.

4.4 Agent Request Timeout on Slow Connections

The self-verifying agent loop can issue up to eight sequential API calls to Gemini. On a reliable connection this completes in fifteen to twenty-five seconds. On slower connections, or when the Cloud Run instance is cold-starting after a period of inactivity, the total latency can exceed sixty seconds, which triggered the frontend's default HTTP timeout and caused the request to fail silently from the user's perspective.

The immediate fix was to extend the frontend's timeout to 120 seconds and add a visible loading indicator that counts elapsed time, so the user knows the request is in progress rather than stalled. The underlying architectural fix — replacing the synchronous request-response model with a server-sent events stream — was designed but deferred, as it required changes to both the FastAPI backend and the frontend fetch layer that exceeded the available time.

4.5 Cross-File Context Contamination

An early version of the extension maintained a single conversation history shared across all open files in the workspace. When a developer worked on a Java file, then opened a Python file and requested test generation without closing the panel, the agent received a message history containing Java code samples, Java package declarations, and Java-specific tool outputs. The result was Python test code containing JUnit annotations and Java import statements.

This was resolved as part of the same architectural change that addressed the Webview state loss problem: by scoping session state to individual file URIs, the Java and Python contexts became completely isolated. The fix also resolved a subtler variant of the problem where the language detection logic was sometimes overridden by the language inferred from earlier messages in a contaminated history.

5. Testing and Validation

Testing was conducted at two levels: functional testing against the metrics defined in the Functional Specification, and an empirical evaluation of the AI component's coverage capabilities using the Problem10 benchmark described in the Research Document.

5.1 Functional Metrics

The following table summarises the results of functional testing against the non-functional requirements defined in the Functional Specification:

Metric	Target	Result	Status
Test generation response time	< 120 seconds	15–25s typical; <60s on cold start	Met
IDE CPU overhead when idle	Minimal impact on VS Code	< 1% CPU, < 50 MB RAM observed	Met
Test file write integrity	No corrupted or partial files	Zero file corruption across all test runs	Met
Extension crash isolation	No IDE crash on backend failure	Verified: backend 500 errors, timeouts, and network loss all handled gracefully	Met
GDPR compliance	No persistent storage of user code	Code sent to backend only for inference; no logging of content in production	Met

5.2 AI Branch Coverage Evaluation

The Research Document presents a detailed evaluation of the AI-generated test strategy against the Problem10 benchmark from the SV-COMP RERS 2012 suite — a Java transpilation of a complex C finite state machine with approximately 60 error-triggering branches. The key results are summarised here for completeness.

The Gemini-generated BFS exploration strategy achieved 76% branch coverage at the bytecode level as measured by JaCoCo, and 86.7% coverage of the core calculate_output method at the source level as measured by OpenClover 4.5.2 (276 real source-level branches). The non-AI baseline for the equivalent C implementation, established using 42 hand-crafted tests with traditional coverage tools, is 76.7%. The AI result is closely aligned with this baseline, suggesting that the generated strategy is competitive with expert-crafted tests on this benchmark.

Two structural reasons for the uncovered 24% were identified through code analysis: combinatorial explosion of the state space caused by the inclusion of a large arithmetic variable in the BFS state

fingerprint, and compound conditional constraints that cannot be simultaneously satisfied given the arithmetic progression of that variable through the state machine. These are genuine limitations of search-based test generation that would require symbolic execution or constraint solving to overcome — capabilities beyond the scope of this project.

5.3 Hallucination Rate Before and After Mitigation

Manual testing before the anti-hallucination measures described in Section 4.1 were implemented found that approximately 40% of generated test files contained at least one ghost file path annotation or invalid import statement that caused immediate compilation failure. After implementing the structured JSON output schema, the regular expression post-processor, and the extended negative system prompting, the failure rate in subsequent manual testing across approximately 60 generation requests dropped to under 5%. The residual failures were confined to edge cases involving deeply nested or non-standard package structures.

6. Reflections and Lessons Learned

6.1 Verification Is the Core Problem, Not Generation

The most important insight from this project is that generating test code is the easy part. A large language model can produce syntactically plausible test code in seconds. Making that code trustworthy — ensuring it compiles, runs, and asserts meaningful behaviour — requires substantially more engineering than the generation itself. This was not anticipated at the start of the project. The original plan allocated most of the backend complexity to the generation prompt; in practice, most of it went to verification, anti-hallucination measures, and output validation.

A consequence of this is that the self-verifying agent loop should have been the first feature designed and implemented, not one added reactively after observing failures. If the project were to begin again, the architecture would start from the question "how do we verify that the output is correct?" rather than "how do we generate output?"

6.2 Platform-Specific Constraints Cannot Be Inferred from Documentation

Several of the most time-consuming bugs in this project — the Webview lifecycle behaviour, the classpath construction challenge, the postMessage serialisation constraints — arose from platform-specific behaviours of the VS Code extension environment that are documented but not prominently advertised. The assumption that VS Code extension development would be broadly analogous to web application development proved to be incorrect in ways that were only discovered through direct experience.

The practical lesson is that any project involving a constrained platform environment (IDE extensions, mobile frameworks, embedded systems) requires a dedicated investigation of that platform's specific constraints before the architecture is finalised. A one-week spike building throwaway prototypes to probe the boundaries of the platform would have prevented several weeks of corrective rework.

6.3 Research Must Be Validated with Prototypes

The Research Document concluded that LangChain was the appropriate framework for the AI backend. This conclusion was based on documentation review and was correct at the level of the framework's stated capabilities. During implementation it became clear that LangChain's linear chain architecture was insufficient for a cyclical, self-verifying workflow, and LangGraph was adopted instead. The switch required rewriting the agent service from scratch but ultimately produced a better architecture.

The lesson is not that the research was wrong, but that it was incomplete: documentation review can establish whether a framework supports a capability in principle, but only a working prototype can establish whether it supports it in the specific way the project requires. Future projects should include a prototyping phase as part of the research deliverable rather than treating the two as sequential.

6.4 User Testing Surfaces Problems That Self-Testing Cannot

Both pieces of feedback from user testing — the missing activity bar icon and the always-enabled generate button — were usability issues that had been invisible during self-testing because the developer

already knew how to use the tool. The missing icon was not noticed because the developer had the extension loaded in development mode, where it appears differently. The always-enabled button was not noticed because the developer habitually selected code before opening the panel.

Even a small amount of user testing with people unfamiliar with the tool's intended workflow would have surfaced these issues earlier. For a developer tool, where the developer-as-author has significantly different mental models from a developer-as-user, external testing is especially important.

6.5 Integration Testing Should Not Be Deferred

Because the frontend depends on the backend and the backend's output depends on the frontend's input format, the two components could not be fully tested in isolation. Integration testing was repeatedly deferred during the project in favour of isolated component development, which meant that several cross-component bugs — including the context contamination issue described in Section 4.5 — were not discovered until late in the development cycle when they were more expensive to fix.

Establishing even a minimal integration test harness early in the project — using a mocked LLM backend that returns predictable responses — would have provided a continuous safety net and would have caught most of the cross-component issues at the point of introduction rather than weeks later.

7. Future Recommendations

The following improvements are recommended for future development, ordered roughly by expected impact relative to implementation effort.

7.1 Button State Management and UI Polish

The most immediate improvement required is disabling the "Generate Tests" button when no code is selected in the editor. Currently the button remains active regardless of the selection state, which leads to confusing empty-context generations. VS Code's onDidChangeTextEditorSelection event can be used to monitor the selection and update the button state in real time. This is a low-effort change with high user experience impact and should be the first item addressed in the next development cycle.

7.2 Streaming Architecture for Agent Progress

Replacing the current synchronous HTTP request with a server-sent events (SSE) stream would allow the frontend to display incremental progress as the agent works through its loop — showing which tool is currently executing and what intermediate results have been returned. This would eliminate the perception of the interface hanging during long generations, reduce timeout failures on slow connections, and give the developer insight into the agent's reasoning process. The change requires modifications to both the FastAPI backend (replacing the standard endpoint with an EventSourceResponse) and the frontend fetch layer.

7.3 Dependency Resolution via Language Server Protocol

Integrating with the Java Language Server Protocol (jdtls) or Python's Pylsp to automatically resolve import dependencies before constructing the AI prompt is the single change most likely to improve the quality of generated tests for real-world codebases. Rather than requiring the developer to manually identify and attach relevant interface files, the extension would transparently resolve the full type graph of the target class and inject it into the context. This is a non-trivial implementation but represents the completion of the F-01 use case as originally specified.

7.4 Session Persistence via globalState

Persisting chat history and specification bindings to VS Code's globalState API would allow developers to resume testing conversations across IDE restarts. The per-file session store already exists in memory; extending it to read from and write to globalState on session create and update respectively is a modest change that would meaningfully improve the experience for developers working on longer-term projects.

7.5 Marketplace Publication with Mandatory API Key Setup

Publishing the extension to the VS Code Marketplace with a first-run walkthrough that requires the user to configure their own Gemini API key would make the tool accessible to a broader audience without the quota management problem that prevented publication during the project. This also has the benefit of generating real-world usage data — both quantitative (request volumes, error rates) and qualitative (through user reviews) — that would provide a more reliable basis for future development priorities than the small-scale informal testing conducted during the project.

8. Conclusions

IntelliTesting delivers a functional AI-driven test generation tool that addresses the central problem it set out to solve: the friction of manual unit test writing for independent developers. All mandatory requirements were met, the most significant discretionary features were delivered, and the system performs reliably in the use cases for which it was designed.

The project's most substantive contribution is its architectural response to the oracle problem. By treating developer-provided specifications as ground truth and generating deliberately failing tests when the implementation contradicts them, the system does more than automate boilerplate — it actively supports the detection of logical defects. The interactive verification mode extends this principle to cases where no specification is available, ensuring that the developer's implicit knowledge of correct behaviour is captured before code is generated.

The development process produced a lesson that is more broadly applicable than the project itself: LLM-based developer tools require as much engineering investment in output verification as in output generation. The gap between a model that can generate plausible test code and a system that developers can trust to generate correct test code is not bridged by better prompts. It is bridged by structured output schemas, executable verification loops, anti-hallucination post-processing, and the kind of systematic debugging that characterises any serious software engineering project.

The limitations that remain — dependency resolution, session persistence, and marketplace publication — have clear implementation paths and do not represent fundamental obstacles. The codebase and architecture produced by this project provide a solid foundation for continued development, and the lessons documented here should inform the next iteration.