Research document

IntelliTesting

Chengyan Song

Dec 2025

SETU

Final Year Project

1. Introduction
2.Related project research
    2.1 Qodo Gen
    2.2 LangChain
3. Project Architecture Research
    3.1 Web Application Front-End Analysis
    3.2 VS Code Extension Front-End Analysis
    3.3 Conclusion
4.Technical Analysis
    4.1 AI LLM(Gemini) capabilities Analysis
        4.1.1 Example 1
        4.1.2 Example 2: Analysis of Consistent Implementation
        4.1.3 Example 3
5. Gemini extreme ability test
    5.1 Experimental Background and Objectives
    5.2 Analysis of Test Generation Strategies
        5.2.1 Scope of Applicability and Limitations
        5.2.2 Core Strategies Implemented
    5.3 Quantitative Analysis of Coverage Data
    5.4 Deep Attribution: Analysis of the 24% Uncovered Branches
        5.4.1 Combinatorial Explosion of the State Space
        5.4.2 Mathematical Constraints in Compound Conditions
        5.4.3 Unreachable Code and Short-Circuit Logic
    5.5 Conclusion
6. Appendix
    6.1 Jira
    6.2 Qodana
    6.3 Benchmark code and AI-generated test cases
References

1. Introduction

This document outlines the research and development undertaken for the Final Year Project (FYP) titled IntelliTesting: An AI-Driven Test Case Generator. As software systems grow in complexity, the cost of manual unit testing has become a significant bottleneck in the Software Development Life Cycle (SDLC). While Large Language Models (LLMs) have shown promise in code generation, integrating them seamlessly into existing workflows remains a challenge.

Project Scope: IntelliTesting focuses on developing a VS Code extension backed by a custom service to automate crucial stages of the software testing process. Unlike traditional standalone tools, this project aims to achieve a "Zero-Interruption Workflow," allowing developers to generate, run, and validate tests without leaving their coding environment.

Document Structure: This report details the architectural decisions (comparing Web Apps vs. IDE Extensions), technical analysis of the Gemini model's capabilities in bug detection, and an empirical "extreme ability test" conducted on complex legacy code (*Problem10*). Finally, it presents the strategic implementation of Agentic AI using the LangChain framework.

2.Related project research

2.1 Qodo Gen

Qodo Gen[1] is defined as a "quality-first" generative AI coding agent platform. It provides developers with comprehensive AI assistance, specializing in unit test generation. The platform supports a wide range of programming languages, including Python, JavaScript, TypeScript, Java, C++, and Go, ensuring broad applicability across

different technology stacks. The tool is built on an Agentic architecture that allows it to execute multi-step tasks and integrate directly within the IDE, making it a critical benchmark for IntelliTesting's goal of achieving a Zero-Interruption Workflow.

Qodo Gen's focus on generating quality tests—including happy paths, edge cases, and rare scenarios—directly informs the functional goals of IntelliTesting's F-02: Intelligent Test Generation.

2. Core Functionality and Architecture

Qodo Gen's architecture validates IntelliTesting's design by demonstrating the feasibility of integrating autonomous AI capabilities directly into the development environment.

2.1 Agentic Architecture and Context Management

- **Agentic Mode and MCP:** Qodo Gen utilizes an Agentic Mode that integrates external tools via the Model Context Protocol (MCP). This protocol allows the AI agent to make decisions, execute commands, and carry out tasks autonomously, confirming the viability of treating CLI and File System operations as AI Tools.
- **Multi-Step Task Execution:** The agent can reason through complex, multi-step tasks, including creating and editing files and running terminal commands (with feedback monitoring). This validates the entire sequential flow of IntelliTesting (F-01 to F-04).
- **Context Quality (RAG):** The system relies on Retrieval Augmented Generation (RAG) for context awareness. Critically, it incorporates quality-guardrails to ensure only relevant and high-quality context is used, preventing poor test generation.

2.2 Test Generation and Execution

Qodo Gen's testing workflow provides concrete implementation examples for IntelliTesting:

- **Comprehensive Coverage:** The system analyzes code behavior to generate test suites covering happy paths, edge cases, and rare scenarios.
- **Customization:** Tests are tailored to the project's style and frameworks, and can be refined through added user context. IntelliTesting can leverage this to implement its Custom Configuration requirement.

- **In-IDE Execution:** Qodo Gen enables running and refining tests directly within the IDE. This model confirms the importance of IntelliTesting's F-04 step for integrated validation.

3. Conclusion and Strategic Takeaways for IntelliTesting

Research into Qodo Gen confirms that IntelliTesting's intended architecture is aligned with leading industry standards for AI coding assistants.

- **Validation of Architecture:** Qodo Gen's reliance on Agentic architecture and the MCP protocol validates IntelliTesting's approach to treating CLI and File System operations as callable tools from the AI backend.
- **UX Reference:** The use of dedicated commands and integrated terminal/chat feedback confirms the efficacy of a non-disruptive, single-trigger workflow within VS Code.
- **Prioritization of Context Quality:** IntelliTesting must adopt Qodo Gen's emphasis on high-quality context collection (RAG and quality-guardrails) during F-01 to ensure the AI generates meaningful, non-trivial tests.

2.2 LangChain

LangChain[2]is a popular open-source framework specifically designed for building applications powered by Large Language Models (LLMs). Its fundamental value lies in providing standardized abstractions and tools that enable developers to:

1. **Connect LLMs to External Data:** Allow the LLM to access context from project files, databases, or documentation.
2. **Implement Complex Reasoning (Chains):** Link multiple LLM calls and logic steps together to solve tasks too complex for a single prompt.
3. **Enable LLM Agency (Agents):** Allow the LLM to plan steps and use external software Tools to interact with the environment.

For the IntelliTesting project, LangChain is the critical backend framework required to implement F-02 (Intelligent Test Generation) and manage the subsequent automated execution flow.

2. LangChain Core Components and IntelliTesting Application

LangChain is structured around key modules that directly solve the technical challenges of integrating AI into the IntelliTesting workflow.

2.1 Prompts and Output Parsers

This is essential for the reliability of the AI output.

- **Function:** Manages the construction, optimization, and template creation for instructions sent to the LLM. The Output Parsers module is used to ensure the model's response adheres to a strict format.
- **IntelliTesting Application:** This module is critical for building the highly structured System Prompt for F-02. The prompt must inject the code, language, target framework, and—most importantly—mandate the precise JSON output schema required by the frontend filename, test_code. This ensures the output is machine-readable and reliable.

2.2 Chains

- **Function:** Provides the architecture to link various components (LLMs, prompts, other chains) into a defined sequence of steps.
- **IntelliTesting Application:** Chains can be used to design the testing workflow. For example, the chain could consist of three steps: 1) Analyze code intent, 2) Generate test assertions, and 3) Validate the output JSON structure before sending the final result to the frontend.

2.3 Retrieval

- **Function:** Handles the loading, indexing, and querying of external data (like project files or documentation) to augment the prompt with specific context.
- **IntelliTesting Application:** While F-01 captures the initial context, the Retrieval component could be used in the future to automatically identify and load dependent files (e.g., interfaces or imported classes) to ensure the AI has the complete context necessary to generate accurate, non-trivial unit tests.

2.4 Agents and Tools

- **Function:** Agents allow the LLM to decide on the next course of action and execute tasks by calling predefined Tools (external software functions).
- **IntelliTesting Application (Zero-Interruption):** Agents are key to enabling the automated execution loop:
    - The F-03 File Writing and F-04 Terminal Execution steps can be abstracted as LangChain Tools FileWriteTool and RunTestTool.

    - The LLM, after generating the code, instructs the Agent: "First, call FileWriteTool with the test code. Then, call RunTestTool with the class name." This autonomous planning is what drives the seamless workflow.

3. Strategic Conclusion and Implementation Recommendation

LangChain is an ideal strategic fit for the IntelliTesting project's architecture, as it fundamentally addresses the challenge of making an LLM interact with a development environment.

- **Implementing the Agentic Flow:** LangChain should be adopted as the core backend framework to facilitate the Agentic approach. This allows the system to move beyond simple text generation to actual task automation and verification.
- **Enhancing Reliability:** Utilizing LangChain's Output Parsers enforces the data contract between F-02 and F-03, directly supporting the project's Reliability requirements by ensuring the AI's output is always structured and parsable.
- **Future-Proofing:** LangChain provides a robust foundation for expanding IntelliTesting's capabilities, such as integrating web searches or complex debugging workflows, which can be added simply by defining new Tools.

3. Project Architecture Research

The IntelliTesting project requires a robust front-end interface to facilitate developer interaction with the underlying AI services. Two primary front-end architectural forms are under consideration: independent web applications (Web Apps) and integrated development environment (IDE) plug-ins, specifically a VS Code Extension.

3.1 Web Application Front-End Analysis

The web application approach requires building a dedicated, full cloud compiler environment within the user's web browser, where the entire software development lifecycle takes place.

Advantages:

1. The complete environment for running, compiling, and testing code is hosted on a remote server. This means the performance of the code execution and testing process is independent of the user's local computer's processing power, instead relying on the scalable cloud infrastructure.

2. Hosting the full environment in the cloud provides a single, central location for all project configurations and data. This inherent centralization simplifies the deployment of common dependencies and makes it straightforward for team leaders to manage access and project environments, which is more suitable for team collaboration and administration.

Disadvantages:

1. To deliver a functional user experience, the project is required to replicate essential IDE features such as syntax highlighting, file tree navigation, and advanced debugging tools. This necessity forces the project to spend significant development time mimicking the functions of modern IDEs like VS Code or IntelliJ IDEA, diverting resources away from the core AI test generation functionality.
2. Because the entire process, from code editing to compilation and testing, is hosted remotely, the complete development and testing workflow is strictly dependent on a stable network connection. This makes offline development impossible for any stage of the process.

3.2 VS Code Extension Front-End Analysis

The VS Code Extension architecture utilizes the existing, powerful features of the VS Code IDE, focusing on integrating the AI service seamlessly into the developer's local environment.

Advantages:

1. The project does not require developing a custom IDE interface; it immediately benefits from VS Code's rich feature set, including its advanced editor, debugging tools, theme support, and extension ecosystem.
2. VS Code is an open-source IDE; this structure provides extensive documentation and tutorials for understanding its internal architecture and accelerating extension development and simplifying maintenance.
3. Only the communication with the backend LLM for test generation requires an internet connection. The time-consuming step of writing the application code can be realized in a local, network-free environment, allowing for offline development of the main codebase.
4. By eliminating the requirement to build a bespoke IDE interface, the development team can concentrate the majority of its effort on creating the core test case

generator logic, thereby maximizing efficiency and improving the quality of the AI's output.

3.3 Conclusion

The core value proposition of the IntelliTesting project is to implement a complete automation cycle: leveraging AI to automatically read local codebase context, generate high-quality test cases, and immediately execute those tests for verification.

A standard Web Application architecture fundamentally fails to deliver this value because it cannot access the user's local filesystem or control the terminal environment, thereby breaking the automation loop. Furthermore, attempting to develop a web application would drastically slow down development efficiency and compromise the quality of the final product, as significant time would be spent trying to replicate the complex features of a modern IDE interface. Therefore, the VS Code Extension is the only suitable architecture.

4.Technical Analysis

1. LangChain: LangChain is a framework for developing applications powered by large language models (LLMs).

LangChain simplifies every stage of the LLM application lifecycle:

Development: Build your applications using LangChain's open-source components and third-party integrations. Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.

Productionization: Use LangSmith to inspect, monitor and evaluate your applications, so that you can continuously optimize and deploy with confidence.

Deployment: Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Platform.[2]

4.1 AI LLM(Gemini) capabilities Analysis

4.1.1 Example 1: Intent-Implementation Mismatch

Scenario: A function identifier *min()* implies an intent to return the minimum value. However, the implementation logic incorrectly returns the maximum value, creating a semantic conflict.

```
int min(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

Test Case Analysis:

- Input: min(3, 8)
- Expected Output: 3 (Based on function intent)
- Actual Output: 8 (Based on faulty logic)

Conflict/Bug Report:

Error Type: Intent-Implementation Mismatch

Conflict: Output 8 fails to match the expected 3 for min(3, 8).

Conclusion: The code implements max logic, violating the function name (min). Correction is required.

The example provided, where the function is named min() but the internal logic actually implements max() (returning the greater of two values), represents a critical class of errors: Intent-Implementation Mismatch. The IntelliTesting AI Backend (F-02) is designed to identify and expose this type of bug, even without a formal specification document.

AI Detection Mechanism

The specific model evaluated, Gemini 2.5 Pro, demonstrates a capability to detect semantic discrepancies through a multi-stage inference process, rather than relying solely on the provided syntax. This process involves:

1. Intent Inference (Heuristics): The model first parses the function signature, specifically the identifier min. By leveraging training data on standard coding conventions and mathematical definitions, it infers the Intended Behavior: the function is expected to return the minimum value of the inputs (e.g., min(3, 8) should yield 3).

2. Semantic Logic Trace: Instead of executing the code, the model semantically traces the conditional logic: if (x > y) return x; else return y;. This analysis correctly identifies that the implementation will consistently return the larger integer, thereby determining that the Actual Behavior corresponds to maximization logic.

3. Conflict Resolution: Upon detecting the divergence between the inferred intent (min) and the analyzed logic (max), the model prioritizes the Intended Behavior (derived from the naming convention) as the "source of truth" for test generation. This enables the generation of assertions that intentionally fail against the buggy code, highlighting the defect.

Test Case Generation Strategy

The following table details the specific test scenarios and assertions generated by Gemini during the experiment. By inferring the correct intent, the model produced the specific assertEquals statements listed in the final column to explicitly expose the bug:

Table 5.1: Test Scenarios and Assertions Generated by Gemini

Test Case Scenario	Input (x, y)	Expected Output (Based on min Intent)	Actual Code Output (Based on max Logic)	AI-Generated Assertion Code
Basic Mismatch	min(5, 10)	5	10	assertEquals(5, min(5, 10));
Reverse Input	min(15, 7)	7	15	assertEquals(7, min(15, 7));

When these generated tests are executed in the Run & Validate (F-04) stage, they will fail. For instance, the assertion assertEquals(5, min(5, 10)) will fail because the faulty code returns 10.

This process demonstrates that IntelliTesting's AI functions as an Intelligent Code Reviewer. By rigorously enforcing semantic naming heuristics through assertion

generation, the system ensures that implementation logic aligns with function naming, delivering high-value bug detection even when a formal specification is absent.

4.1.2 Example 2: Analysis of Consistent Implementation

Scenario: Consider a case where the external business requirement specifies a function to find the maximum value, but the developer mistakenly implements a correctly functioning minimum function named `min`. This creates a scenario where the code is internally consistent but fails to meet external requirements.

```
int min(int x, int y) {
    if (x > y) {
        return y;
    } else {
        return x;
    }
}
```

AI Analysis and Testing Strategy

When the specification is absent, the AI loses its "source of truth" and must rely entirely on coding conventions and naming heuristics to infer the function's purpose.

1. **Intent Inference:** The AI interprets the function name min as a clear indication that the intended behavior is to return the minimum of the two inputs.
2. **Code Consistency Check:** The AI analyzes the implementation: if (x > y) return y; else return x;. The logic correctly returns the smaller value.
3. **Test Generation:** Since the inferred intent (min) matches the actual implementation (min), the AI generates tests designed to validate this internal consistency.

In this scenario, where the specification is unknown, the AI successfully fulfills the role of a Code Consistency Validator.

- **Test Outcome:** All generated tests will PASS when executed in the F-04 stage.
- **Limitation:** The AI correctly validates that the min function works as a min function. However, because the system's external requirement for a max function was not provided to the AI, the actual business logic error is not detected. The test only confirms the code's consistency with its name, failing to detect the functional requirement violation. This highlights the necessity of providing clear requirements to the AI for detecting high-level functional bugs.

4.1.3 Example 3: Implementation Bias in Boundary Testing

The code under test is as follows:

```
public boolean approveLoan(int income, int creditScore
                                      , boolean hasGuarantor) {
    if (income < 0 || creditScore < 0) {
      throw new IllegalArgumentException();
    }

    if ((income > 40000 && creditScore > 650) || hasGuarantor) {
      return true;   // Loan approved
    } else {
      return false;  // Loan rejected
    }
  }
}
```

Test Case Generated (from Gemini, Prompt: "Generate JUnit test cases for this code that achieve the highest possible branch coverage"):

```
class approveLoanTest {
    private final approveLoan loanApprover = new approveLoan();
    @Test
    void testApproveLoan_HighIncomeAndCreditScore() {
        assertTrue(loanApprover.approveLoan(50000, 700, false));
    }
    @Test
    void testApproveLoan_WithGuarantor() {
        assertTrue(loanApprover.approveLoan(30000, 600, true));
    }
    @Test
    void testApproveLoan_LowIncomeAndCreditScore() {
        assertFalse(loanApprover.approveLoan(30000, 600, false));
    }
```

```java
    @Test
    void testApproveLoan_HighIncomeLowCreditScore() {
        assertFalse(loanApprover.approveLoan(50000, 600, false));
    }
    @Test
    void testApproveLoan_LowIncomeHighCreditScore() {
        assertFalse(loanApprover.approveLoan(30000, 700, false));
    }
    @Test
    void testApproveLoan_BoundaryLow() {
        assertFalse(loanApprover.approveLoan(40000, 650, false));
    }
    @Test
    void testApproveLoan_BoundaryHigh() {
        assertTrue(loanApprover.approveLoan(40001, 651, false));
    }
    @Test
    void testApproveLoan_BoundaryIncomeHigh() {
        assertFalse(loanApprover.approveLoan(40001, 650, false));
    }
    @Test
    void testApproveLoan_BoundaryCreditScoreHigh() {
        assertFalse(loanApprover.approveLoan(40000, 651, false));
    }
}
```

Result:

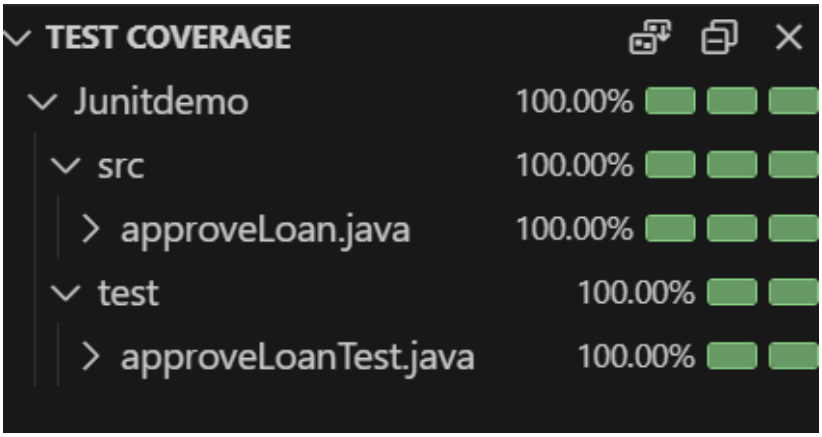


Figure 4.1: *The approveLoan method and the resulting 100% coverage report, demonstrating how Gemini captures the full logic of the current implementation.*

4.1.3.1 Critical Analysis: The Paradox of Implementation Bias

The Risk (The "Why it is dangerous"): While the generated test suite achieves 100% branch coverage, it relies on a fundamental assumption: that the current implementation is correct. This creates a risk of Implementation Bias. For example, if the income threshold in the source code was erroneously set to 4000 instead of 40000, Gemini would generate a test case expecting approval for 4001. In this scenario, the test merely mirrors the bug rather than exposing it, creating a false sense of security.

The Value (The "Why test then?"): Given this limitation, the primary value of AI-generated tests from existing code is not necessarily to find *existing* logical bugs, but to serve as a Regression Safety Net for future refactoring.

- Refactoring Support: By capturing the current behavior of the legacy code into a test suite, developers can refactor or optimize the code (e.g., cleaning up the approveLoan logic) with the confidence that the behavior remains unchanged.
- Back-to-Back Testing: These tests act as a baseline. Any deviation in future iterations will cause these tests to fail, alerting the developer to unintended behavioral changes.

approveLoan.java

```
1.  package com.cm;
2.
3.  public class approveLoan {
4.      public boolean approveLoan(int income, int creditScore, boolean hasGuarantor) {
5.          if (income < 0 || creditScore < 0) {
6.              throw new IllegalArgumentException();
7.          }
8.
9.          if ((income > 40000 && creditScore > 650) || hasGuarantor) {
10.             return true;    // Loan approved
11.         } else {
12.             return false;   // Loan rejected
13.         }
14.     }
15.     }
```

Figure 4.2: *Source code of the* `approveLoan` *method with coverage indicators. The green diamonds in the gutter confirm that all conditional branches were executed by the Gemini-generated tests.*

Critical Limitation: The Risk of Implementation Bias

While the generated test suite achieves 100% branch coverage, it highlights a fundamental risk in relying solely on code-to-test generation: Implementation Bias.

As noted in the generated assertions (e.g., `assertTrue`), the AI assumes the current implementation of `approveLoan` is the "source of truth." If the original logic contained a functional error—for instance, if the income threshold was accidentally coded as `4000` instead of `40000`—Gemini would generate a test case expecting the code to approve a loan with an income of `4001`.

Consequence: The generated test would PASS because it mirrors the faulty logic, creating a false sense of security. Conclusion: Without external specifications (such as Javadoc or requirement documents) to cross-reference, the AI acts as a Regression Guard (ensuring behavior doesn't change) rather than a Correctness Validator (ensuring behavior is right). This underscores the necessity of the "Human-in-the-loop" approach in the IntelliTesting workflow.

5. Gemini extreme ability test

5.1 Experimental Background and Objectives

This chapter presents an empirical evaluation of Gemini's ability to generate "limit testing" scenarios for legacy code characterized by high cyclomatic complexity and strong state dependency. The subject of this experiment is the *Problem10* class.

Origin of the Experimental Subject: The *Problem10* class is not a native Java benchmark. It was originally a standard C benchmark that was transpiled into Java by Gemini for the purpose of this experiment. This process involved significant architectural adaptation rather than a simple rewrite; notably, Gemini converted the original C global variables into Java static fields to preserve state persistence. This specific AI-processed structure introduces unique challenges, such as static state pollution, which are addressed in the subsequent testing strategy.

The resulting class is a typical implementation of a finite state machine (FSM) containing complex arithmetic operations and approximately 60 potential error triggers (verifyError).

The core objective of this experiment is to verify whether the test code generated by Gemini can:

1. Comprehend and manipulate hidden global states (specifically variables a1, a10, and a19).
2. Trigger deep logical branches through specific input sequences (integers 2 through 6).
3. Achieve high code coverage and identify logical areas that remain uncovered, along with the reasons for these omissions.

5.2 Analysis of Test Generation Strategies

5.2.1 Scope of Applicability and Limitations

It is important to acknowledge two critical characteristics of this generation strategy that define its specific utility:

- Dynamic Generation and Determinism: Unlike traditional static unit tests, the tests generated by Gemini for this experiment involve dynamic state exploration. This implies that the test suite is not a fixed set of assertions; rather, it is a heuristic search process. Consequently, the achieved code coverage may vary slightly between execution runs due to the non-deterministic nature of the exploration depth and pathing.
- The Oracle Problem and Regression Value: As formal specifications for Problem10 are absent, the generated tests cannot validate functional correctness (i.e., distinguishing whether a calculation result is "correct" or "incorrect"). Therefore, the utility of these tests is strictly defined for:
    - Error State Detection: Identifying inputs that trigger deep logical faults or exceptions (e.g., reachable verifyError states).
    - Back-to-Back Testing: Serving as a "safety net" for regression testing. The generated behaviors act as a baseline snapshot, allowing developers to refactor legacy code while ensuring that the internal state transitions remain mathematically identical to the original version.

5.2.2 Core Strategies Implemented

An analysis of the generated Problem10Test.java reveals that Gemini did not rely on simple random fuzzing. Instead, it demonstrated advanced white-box testing strategies adapted to the specific architecture of the legacy code:

1. State Reset via Reflection (Handling Translation Artifacts)

Instead of a chosen testing strategy, this mechanism addresses a specific architectural side-effect of the C-to-Java transpilation. The original C global variables were converted into static fields in the Problem10 Java class (e.g., a1, a10). To prevent state pollution between test iterations—where one test's modifications to a static variable would affect subsequent tests—Gemini generated a @BeforeEach method utilizing Java's Reflection API. This approach forcibly resets private static fields to their initial values, ensuring a clean state for each execution.

2. Algorithm-Driven State Space Exploration (BFS)

Rather than blind random testing, Gemini implemented a Breadth-First Search (BFS) algorithm within the testExploreStateSpaceForBugs method.

- Mechanism: It maintains a Queue<List<Integer>> to store input histories.
- Process: The algorithm retrieves a history sequence from the queue, resets the state (using the reflection method above), replays the history to restore the system state, and then attempts to append new inputs (integers 2-6). This systematic approach allows it to hunt for bugs by exhausting short sequences before moving to deeper logic.

3. State Fingerprinting (Cycle Detection & Pruning)

To support the BFS strategy, Gemini implemented a mechanism to uniquely identify and track the internal state of the Finite State Machine (FSM).

- Identification: It identified that the static variables a10, a19, and a1 collectively represent the unique state of the system.
- Serialization: To effectively detect infinite loops, Gemini generated a helper method, getCurrentStateFingerprint(). This method serializes the current values of these variables into a composite string key (formatted as "val_a10|val_a19|val_a1").
- Pruning: By storing these keys in a visitedStates set, the algorithm verifies if a state has been encountered before, allowing it to prune redundant branches and prevent infinite loops.

5.3 Quantitative Analysis of Coverage Data

Test Execution Volume: The JUnit framework reports 4 test methods executed (as per the build log). However, these methods function as entry points for the dynamic BFS algorithm. Internally, the algorithm generated and evaluated approximately 50,000 unique input sequences (reaching the configured maxStates threshold) to achieve the reported coverage.

According to the JaCoCo coverage report (index.html), the experimental results are as follows:

- Instruction Coverage: 94% (3,660/3,878)
  This high value indicates that the input sequences generated by Gemini successfully executed the vast majority of code lines.
- Branch Coverage: 76% (1,455/1,902)
  While the performance is robust, 24% (447 branches) remained uncovered.
- Gap Analysis:
  Although instruction coverage is high, the discrepancy in branch coverage (an 18% drop) highlights Gemini's limitations in handling "limit boundary conditions." While the code paths were executed, not all boolean logic permutations (True/False) were fully traversed.

5.4 Deep Attribution: Analysis of the 24% Uncovered Branches

By comparing the logic in Problem10.java with the testing strategy, a detailed inspection of the source code alongside the coverage report indicates that the uncovered branches are primarily concentrated in areas involving specific arithmetic boundaries and state combinatorial explosions. The specific reasons are analyzed below:

5.4.1 Combinatorial Explosion of the State Space

The core difficulty of Problem10.java lies in the state variable a1, which is a large integer that undergoes non-linear arithmetic changes after every input (e.g., a1 = (((a1 + -15535) - 211896) / 5)).

- Side Effects of the Fingerprint Strategy: Gemini included the highly volatile variable a1 as part of the state definition (a10|a19|a1). * The Problem: Because a1 is a large integer that changes value with almost every arithmetic operation, the system treats every slight numerical variation as a completely new and unique "State".
- The Result: Even if the program is logically stuck in the same code block, the BFS algorithm believes it is discovering new territory simply because the value of a1

changed. This floods the search queue with numerically distinct but logically redundant states, causing the algorithm to hit the maxStates limit (50,000) before it can explore deeper branches.

5.4.2 Mathematical Constraints in Compound Conditions

The uncovered branches are mainly located in the error checking logic of Phase 1. For example:

```
if((((((a10==4) && (a12==0)) &&   ((38 < a1) && (218 >= a1)) ) && (a4==14)) &&
(a19==8))){
    verifyError("error_57");
}
```

To trigger this branch, the following must be satisfied simultaneously:

1. a10 == 4
2. a19 == 8
3. a1 must fall precisely within the interval **(38, 218]**.

Gemini's BFS algorithm can effectively navigate the state transitions of a10 and a19. However, it cannot reverse-solve arithmetic equations to deduce the specific input sequence required to land a1 in such a narrow interval. Gemini relies on search heuristics rather than Symbolic Execution or constraint solving. As the value of a1 grows to millions, the probability of it falling back into a small range like [-13, 218] is statistically negligible, causing the relevant if branch to evaluate to False and omitting the verifyError block.

5.4.3 Unreachable Code and Short-Circuit Logic

Although a12 appears frequently in conditional checks (e.g., a12==0), it is initialized to 0 and never modified in the provided code.

- In this instance, while a12==0 is always true, it is often coupled with complex a1 interval conditions (e.g., a1 <= -13) in "short-circuit logic." If the a10 state matches but the a1 interval does not, the line is technically scanned (counting toward instruction coverage), but the specific sub-branch combination of the compound boolean expression is never evaluated as true, resulting in a loss of branch coverage.

5.5 Conclusion

This experiment demonstrates that Gemini possesses superior architectural understanding capabilities. It successfully identified and simulated complex environmental setups (such as resetting static states via reflection) and constructed a logical algorithm (BFS) to hunt for bugs.

However, regarding Limit Case Generation, Gemini exhibits distinct limitations:

1. Lack of Mathematical Solving Capability: When facing path constraints heavily dependent on arithmetic (such as precise integer intervals), Gemini, relying on search/fuzzing approaches, struggles to generate precise solutions and is easily diluted by the massive numerical space.
2. Trade-off Between Resources and Depth: To protect memory resources (limiting the state count to 50,000), the test code written by Gemini was forced to sacrifice search depth when dealing with state machines containing large integer variables, leaving deep logic uncovered.

6. Appendix

The following sections (7.1, 7.2) detail technologies that were investigated during the initial research phase. However, they were subsequently excluded from the final scope because the project's functional requirements have evolved and changed since the project's inception.

6.1 Jira

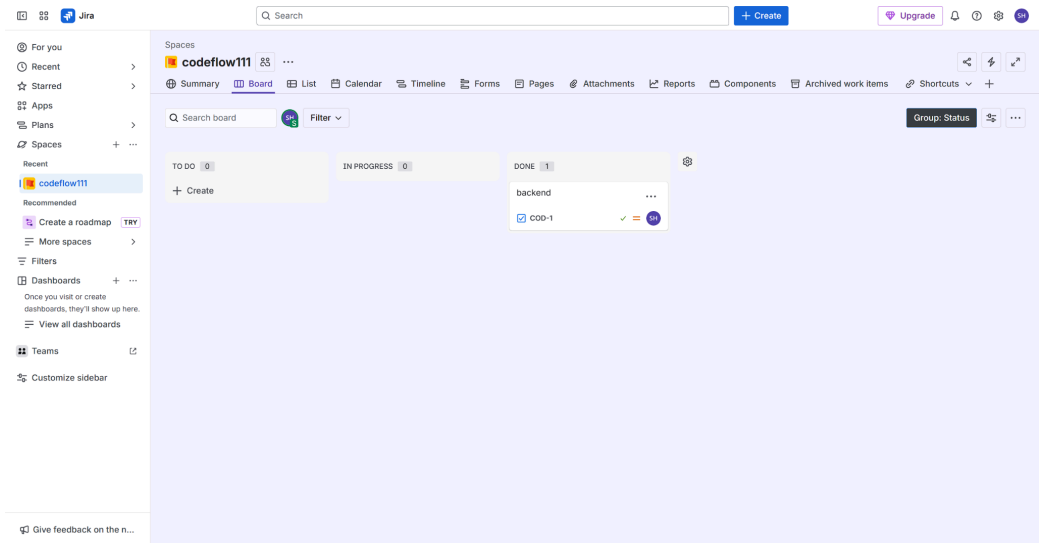
Jira[3] is a highly structured and customizable version of a "to-do list" application, designed as a powerful project and task tracking tool that helps teams plan, execute, and review their work. It not only tells a team "what to do," but also precisely defines "how to do it," making the entire workflow completely transparent. This structure makes Jira particularly suitable for projects requiring team collaboration and complex processes, especially within the software development lifecycle.

Jira's main features can be summarized into four key areas, which provide a comprehensive framework for managing projects from conception to completion.

1. **Create and Manage Tasks** In Jira, any unit of work—whether it's a software bug, a marketing campaign task, or a design requirement—is called an "issue." Each

issue can be assigned a title, description, assignee, due date, and priority, ensuring all necessary information is centralized and clear.

2. **Define the Workflow** One of Jira's most powerful capabilities is its customizable workflows. Teams can define the exact steps a task must go through from start to finish, such as To Do -> In Progress -> Pending Review -> Completed. This standardization ensures that everyone on the team follows the same operational procedures.
3. **Visualize Work Progress** Jira visually displays all tasks as cards on a Kanban or Scrum board. This allows team members to clearly see the status of every task, who is responsible for it, and which tasks may be blocked. This visualization is invaluable for facilitating daily stand-up meetings and tracking overall progress.



4. **Reports and Analysis** The platform can automatically generate various analytical charts and reports. These include pie charts, which can analyze work distribution among team members, and velocity charts, which assess a team's productivity over time. These tools help teams retrospectively analyze their performance and improve future work efficiency

6.2 Qodana

JetBrains Qodana[4] is a code quality platform that provides server-side static analysis of source code. "Its primary technical objective is to decouple the powerful inspection engine from JetBrains Integrated Development Environments (IDEs) and make it executable in any automated environment, particularly within a Continuous Integration/Continuous Deployment (CI/CD) pipeline."[5] This allows for consistent

code quality enforcement by applying the same analysis rules both in the local development environment and in the automated build process.

Qodana's core functionality is achieved through several key technical implementations, as detailed in its official documentation :

- **IDE-Powered** Analysis Engine
    - **Function:** This feature performs static code analysis using the exact same rule set and engine as JetBrains IDEs (like IntelliJ IDEA or PyCharm).
    - **Implementation:** The core inspection engine, which is normally integrated into the IDE's graphical user interface, has been packaged to run as a headless command-line tool. It programmatically parses the source code to build an Abstract Syntax Tree (AST), then applies hundreds of pre-built inspection rules to this tree to find potential bugs, style violations, and performance issues. The results are output in a structured format, such as SARIF.
- **CI/CD** Integration and Quality Gates
    - **Function:** This feature allows Qodana to be executed automatically as part of a CI/CD workflow and to fail the build if the code does not meet pre-defined quality standards.
    - **Implementation:** Qodana is distributed as a Docker image. This containerization allows it to be platform-agnostic. Integration is achieved by adding a step to a CI/CD configuration file (e.g., a GitHub Actions workflow .yml file) that pulls and runs the Qodana Docker image. "Quality gates" are implemented through a qodana.yaml configuration file where a user can specify failure conditions (e.g., failThreshold: 10). The Qodana tool reads this configuration and, if the number of detected issues exceeds the threshold, it exits with a non-zero status code, which signals the CI/CD system to fail the build.
- **Baseline Analysis**
    - **Function:** This feature allows the tool to only report new issues introduced in a code change, ignoring problems that already exist in the main codebase.
    - **Implementation:** On its first run against a branch, Qodana performs a full scan and generates a report file that serves as a "baseline" snapshot. On subsequent runs (e.g., on a feature branch), it performs another full scan. The implementation then involves a computational comparison between the new scan results and the baseline file to produce a delta. Only the issues

present in this delta are reported, which prevents developers from being overwhelmed by legacy technical debt.

6.3 Benchmark code and AI-generated test cases

Benchmark code: Problem10.java[6]

Gemini generate test case: Problem10Test.java

Test coverage report:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Lines	Methods
Problem10	218 of 3,878	**94 %**	447 of 1,902	**76 %**	432	956	453	5
Total	218 of 3,878	**94 %**	447 of 1,902	**76 %**	432	956	453	5

References

[1] *Qodo*. https://github.com/qodo-ai/qodo-cover

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

[3] *Jira Software*. Atlassian. (2025). Research fromhttps://www.atlassian.com/software/jira

[4] *Qodana*. JetBrains. Research from https://www.jetbrains.com/qodana/

[5] *Qodana Documentation*. JetBrains. Retrieved from www.jetbrains.com/help/qodana

[6] *Benchmark Code. gitlab.* *https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/testcomp25/c/eca-rers2012/Problem10_label03.c?ref_type=tags*