

WHITEPAPER

CT's Analysis-Enriched AI Test Generation

Document Information

Document

Distribution date December 5, 2025

Technical Support

Email help@suresofttech.com

Phone +82-31-606-2000

Legal Notice

© 2025. Suresofttech Inc. All rights reserved.

Proprietary and Confidential This material is proprietary to Suresofttech Inc. It contains trade secret and confidential information which is solely the property of Suresofttech Inc. This material is for client's internal use only. This material shall not be used, reproduced, copied, disclosed, and transmitted, in whole or in part, without the express consent of Suresofttech Inc.

Reproduction and distribution without permission from Suresofttech Inc. is prohibited.

For suggestions or inquiries regarding this document or product, please contact Suresofttech Inc.

Some images included in this document may differ slightly depending on the version. The image corresponding to the function being described is identical to the version specified in the document information.

Table of Contents

1	Overview	4
2	Limitations of AI Test Generation	6
	Conventional Generation Approach	6
	Limitations in C/C++	6
	Solution	8
3	Analysis-Enriched AI Test Generation	9
	Project Information Collection and Analysis	9
	Context-Based Prompt Design	10
	Feedback-Based Improvement Loop	11
	Performance	14
4	CI/CD Integration of AI Test Generation	17
	CI/CD-Based Automation Process	17
	Practical impact	18
5	Conclusion and Future Plans	19
	Expected value	19
	Future Plans	19

1 Overview

Large Language Model (LLM) technology is transforming software development. AI now automates test code generation, extending beyond code writing and debugging. A 2024 GitHub report, "AI in Software Development," indicates that 98% of developers worldwide use AI when writing tests. AI has become a critical technology that determines both development quality and speed.

However, this adoption does not apply equally to all programming languages. Generating tests with AI, particularly for C/C++ projects, remains challenging. C/C++ involves complex language constructs such as macros, preprocessor conditions, templates, namespaces, and pointers. In addition, execution results can be heavily affected by external libraries and build options. To generate C/C++ tests accurately, AI must understand not only the code but also the project's build environment and dependency relationships. Chapter 2 discusses these limitations in detail.

To address this challenge, CT has developed a new AI-driven test generation approach. CT is an automated unit and integration testing tool for C/C++ developed by Suresofttech. It supports capabilities required in mission-critical industries, including coverage measurement, standards compliance, and automated checking.

CT has achieved TÜV certification for the following functional safety standards:

Table 1. Domain-Specific Certification Standards

Domain	Certification Standard
General Industry	IEC 61508-3:2010
Automotive	ISO 26262-8:2018
Medical Devices	IEC 62304:2015
Railway	EN 50128:2011/A2:2020
Nuclear Power	IEC 60880:2006

With this certification, CT can be utilized as a validation tool in development environments demanding the highest safety integrity levels, such as SIL 4 and ASIL D.

Based on this proven track record, CT has developed the Analysis-Enriched AI Test Generation feature, which combines AI and analysis technologies. This feature is designed to allow AI to analyze code structure and autonomously improve test quality.

The core of Analysis-Enriched AI Test Generation involves two main aspects:

First is the delivery of accurate analysis information. This entails transmitting essential information for test generation, such as functions, headers, macros, and build options, to the AI.

Second is the feedback loop. It automatically improves quality by re-feeding the compilation and execution results of tests back into the AI. This process addresses two situations:

- If a compilation error occurs, it automatically modifies the test to reflect the error.
- If compilation succeeds but coverage is insufficient, it identifies uncovered code and adds tests.

CT has extended this feature across the entire development pipeline, creating an environment where test generation, building, execution, and coverage measurement continuously cycle. By integrating with CLI and CI/CD environments, seamless AI-driven test quality improvement can be performed throughout the development process.

AI test generation technology can be utilized as a core feature for managing quality in real-world development processes. Subsequent chapters will detail the limitations of existing AI test generation and how CT's analysis-enriched approach addresses them.

2 Limitations of AI Test Generation

Conventional Generation Approach

Generally, when generating test code with AI, a portion of the code related to the function under test is provided to the AI. This includes function signatures, comments, and some code blocks. AI generates test code based on this limited information.

Users must repeatedly confirm compilation success and fix errors. The procedure is as follows:

1. Input function code and request test generation from the AI.
2. The AI outputs test code.
3. The user directly copies the resulting code, then compiles and executes it.
4. If an error occurs, they manually fix it or request it from the AI again.

In this workflow, AI serves merely as a convenience tool that substitutes for manual test code writing. It does not fully utilize AI's potential, such as enhancing test quality or automatically correcting errors.

Limitations in C/C++

Limitations of Information Provided to AI

AI bases its understanding only on the information provided. In C/C++ projects, information such as preprocessor macros, header dependencies, build options, and platform flags is often distributed outside the target source file. If this information is not transmitted, AI cannot accurately identify which headers to include, which macros are active, or where types are defined.

This limitation stems from two structural factors:

- **LLM's input size limitation (context window):** The entire project cannot be input at once.
- **Difficulty in extracting accurate analysis information:** Information such as header inclusion relationships, preprocessor environments, and build settings can only be accurately identified through compiler-level analysis, making extraction itself difficult.

Consequently, AI generates tests using only fragmented information without the full context, leading to compilation errors when building tests, or a high likelihood of generating tests that are irrelevant to the actual operation even if they execute.

Problem of Inability to Confirm Generation Results

Since LLMs are text generation models, they have structural limitations in confirming their generated results. To ascertain whether the generated code actually compiles and executes, integration with external tools is necessary. Without such integration, AI cannot autonomously confirm the correctness of its generated output.

These limitations become even more pronounced in C/C++. Due to a lack of necessary information, the probability of errors is high, and error causes existing outside the code, such as missing headers, macro conditions, and type mismatches, frequently occur. AI finds it difficult to accurately identify the root causes solely from error messages, and it also lacks methods to correct them autonomously. Even if a user corrects an error and requests again, similar errors can recur if AI fails to understand the previous context.

This leads to AI generating tests with the following errors:

- Incorrect variable declarations and initialization errors
- Missing includes
- Mismatched type definitions
- Abnormal test scenario configurations

Consequently, this makes AI-generated tests unreliable and fosters the perception that utilizing AI for test generation is difficult in complex languages like C/C++.

Failure Case: Ninja Build System

An experiment conducted targeting build.cpp in the open-source build system project Ninja clearly illustrates this problem.

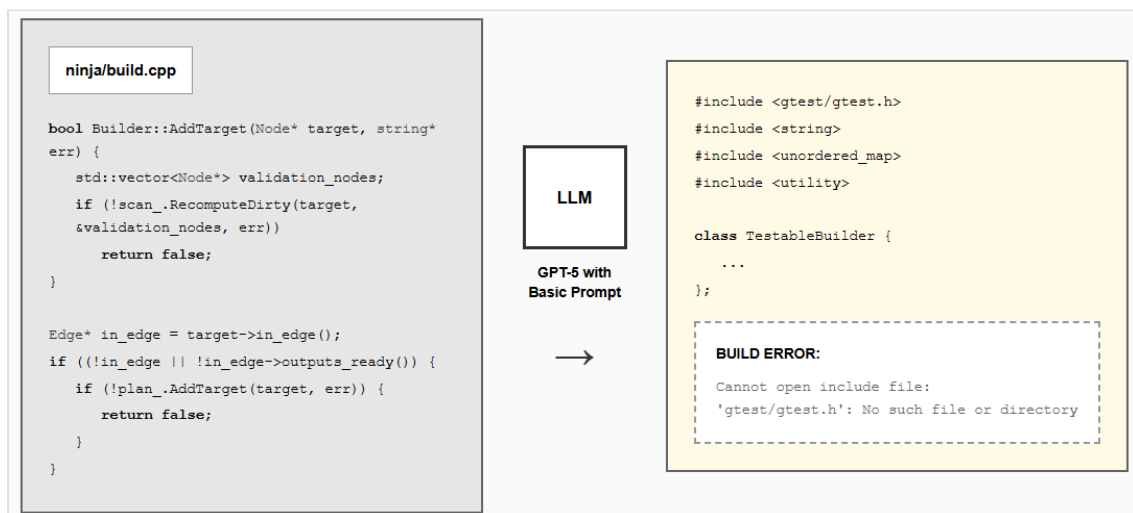


Figure 1 Conventional AI Test Code Generation Results

AI inappropriately included headers, unaware of the gtest library's configuration within the project, and attempted to instantiate an abstract class without understanding its structure. This was due to AI's complete failure to comprehend the project's build environment.

Solution

The core of the problem is that the information provided to AI is limited, and generated results cannot be automatically confirmed or corrected. As C/C++ projects grow in size, related information such as headers, macros, and build options increases exponentially. Even though modern LLMs have large context windows, inputting all code and information is not cost-effective. Furthermore, as the input becomes longer, LLMs tend to miss information located in the middle (information loss, also known as the "Lost in the Middle" phenomenon), which reduces accuracy. In security-critical, mission-critical domains, small models in on-premise environments are often used instead of large cloud models. Small models have smaller context windows, thus further limiting the amount of information that can be input.

CT addresses this problem through the following three approaches:

- **Accurate and Concise Information Provision:** Instead of inputting the entire code, it selectively gathers only essential information required for test generation and delivers it in a refined form.
- **Providing an Interface for AI to Request Necessary Information:** Instead of transmitting all information at once, it enables AI to autonomously determine and request the necessary information through MCP (Model Context Protocol), a standard interface.
- **A Loop to Check Outputs and Provide Feedback:** It actually compiles and executes the generated tests to confirm errors, and feeds those results back to the AI for correction.

When these conditions are met, AI can understand the context of C/C++ code more accurately and improve quality based on test results.

CT's Analysis-Enriched AI Test Generation has implemented all these solutions. It extracts accurate information using code analysis technology and delivers it to the AI, progressively enhancing test quality through a feedback loop.

3 Analysis-Enriched AI Test Generation

This chapter explains how CT's Analysis-Enriched AI Test Generation operates. The core objective is to provide accurate information, enabling AI to understand the project and iteratively improve test quality until its coverage goal is reached.

Project Information Collection and Analysis

CT delivers information collected in two stages to the AI.

Build Environment Information Collection

CT captures the actual software build process. If a build environment is not available, CT identifies project configuration information through its own analysis and user input. During this phase, the following information is gathered:

- **Compilation configuration:** compiler options, optimization level, build flags
- **File dependencies:** header file inclusion relationships and paths
- **External libraries:** external libraries and dependencies used by the project

Code Structure Analysis

Using build environment information as input, CT performs analysis with a commercial compiler-level C/C++ parser. Through this analysis, the following information is extracted:

- **Function Information:** signature details such as function names, parameters, and return types
- **Variable Information:** types and scopes of global and local variables
- **Macro and Preprocessor Information:** preprocessor directives such as `#define` and `#ifdef`
- **Function Relationships:** call graphs detailing which functions call which
- **Execution Flow:** control flow structures within functions (branches, loops, etc.)

Based on the analysis results, probes for coverage measurement are inserted at precise locations in the code to measure precisely.

With the information collected in this way, AI generates tests with an understanding of the full project context, rather than relying on isolated code fragments.

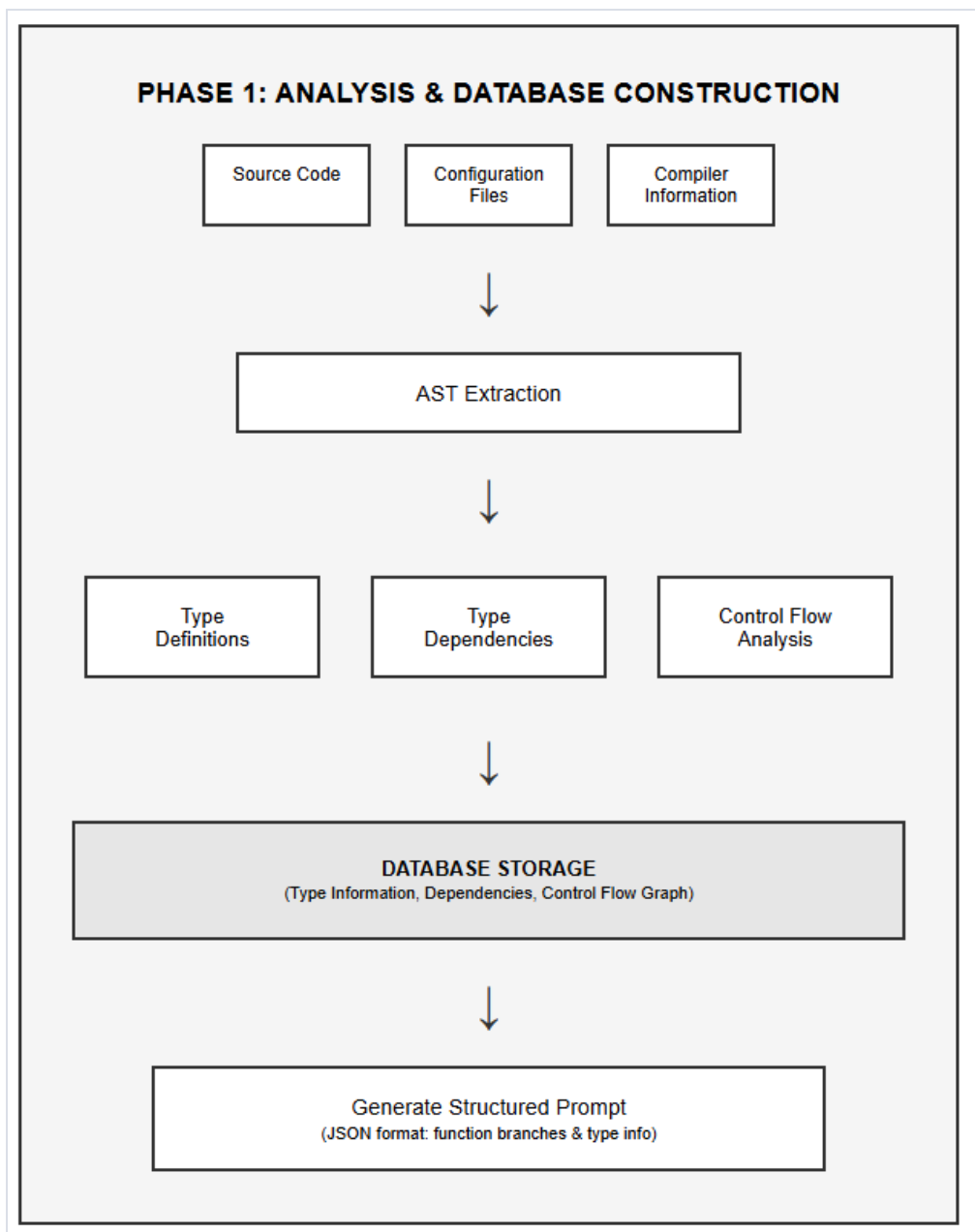


Figure 2 Project Information Collection Process

Context-Based Prompt Design

The collected information becomes the core input material for the LLM. This includes refined function signatures, types, macros, build options, and branch information, which are then transmitted to the AI.

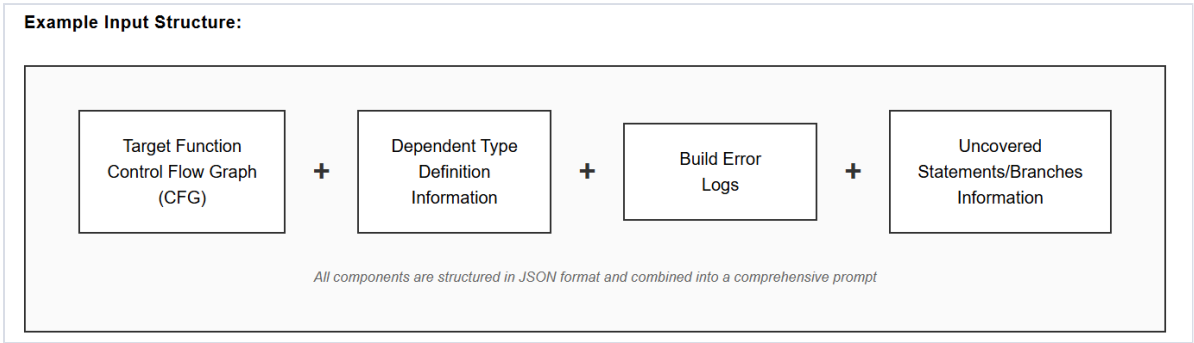


Figure 3 Context-Based Prompt Structure

The main components included in the prompt are:

Table 2. Context-Based Prompt Configuration

Component	Content
Control Flow Graph (CFG)	Branches and execution paths within a function
Type Information	Structs, typedefs, enums, etc., used by the function
Build Error Log	Compilation errors from the previous attempt
Uncovered Statement/Branch Information	Untested code paths

This information is updated with each iteration of the feedback loop, enabling AI to generate more accurate tests.

Feedback-Based Improvement Loop

CT's Analysis-Enriched AI Test Generation continuously improves test quality not through simple repetition, but through a feedback-based improvement loop.

This process proceeds in four stages:

Step 1: AI Test Code Generation

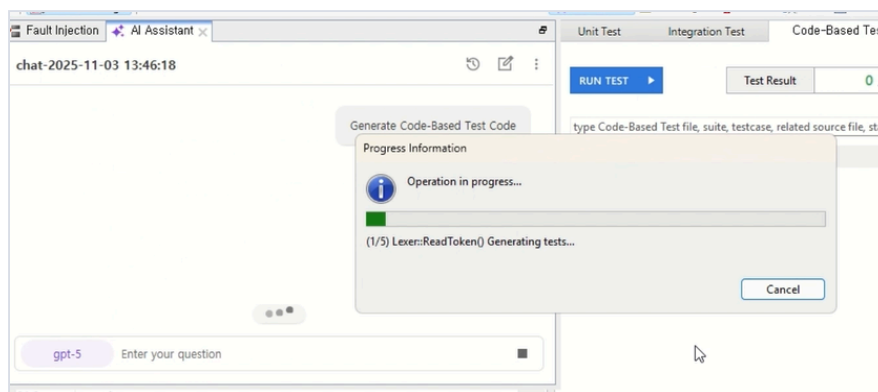


Figure 4 Test Code Generation Screen

The LLM generates the initial test code using the information provided by CT (e.g., function signatures, types, macros, build options, and branching information).

Step 2: Compilation and Execution

CT builds and runs the generated test code in the host environment. It confirms the success of compilation and the test execution results, which are used as feedback data for the next step.

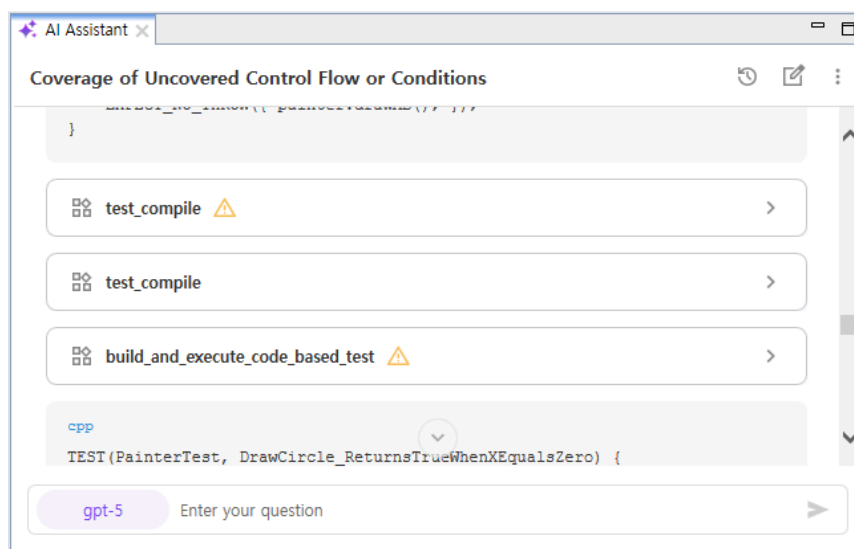


Figure 5 Feedback Collection Screen

Step 3: Feedback Collection

After test execution, CT collects compilation errors and coverage results. Instead of transmitting the entire build log as-is, only essential information is refined to enable AI to accurately identify the root causes of errors and uncovered paths.

Step 4: Test Modification and Regeneration

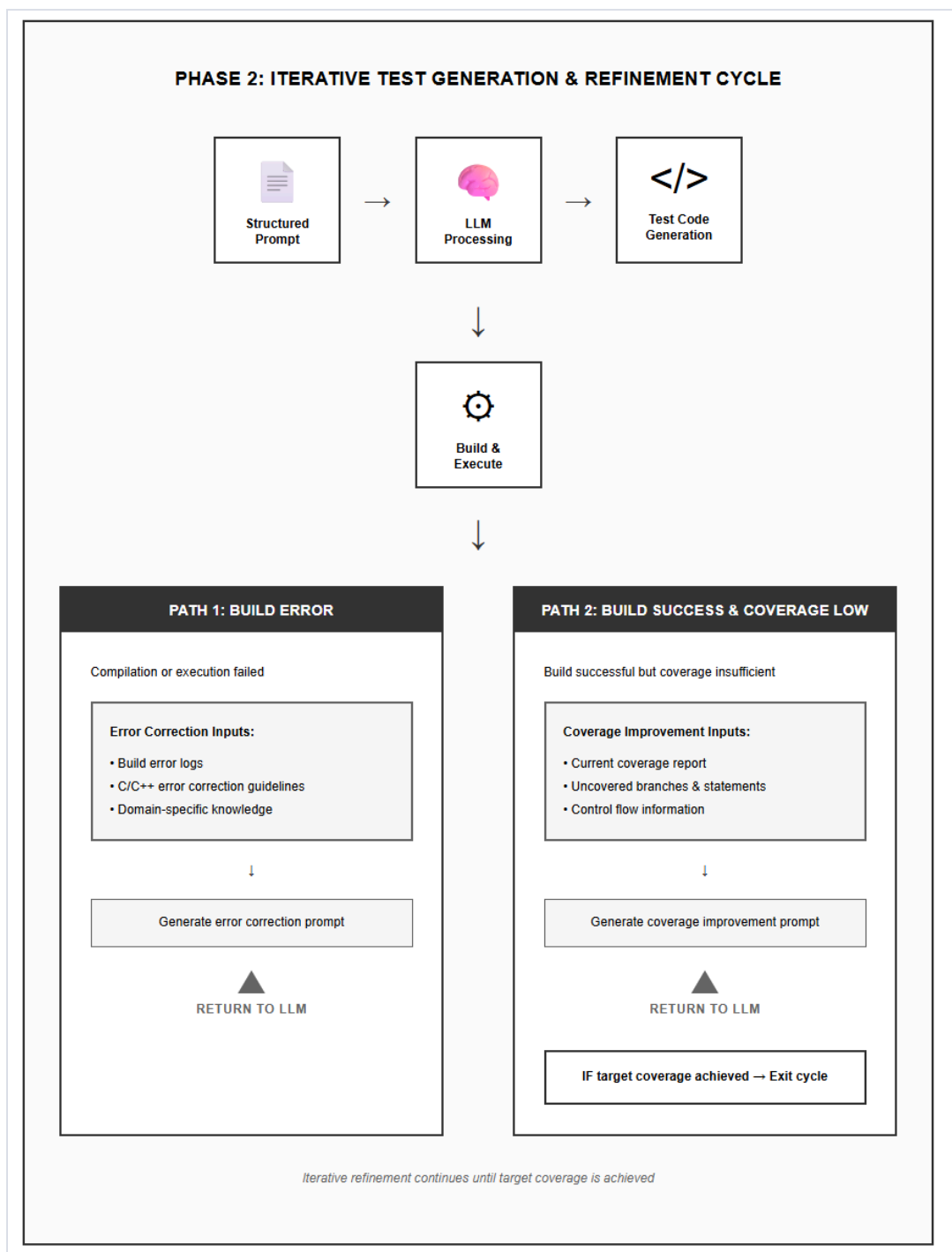


Figure 6 Test Modification and Regeneration Process

Based on the refined feedback, AI modifies existing tests or generates new ones. The process proceeds in two directions, depending on the feedback results:

In Case of Build Error (PATH 1: BUILD ERROR)

If compilation or execution fails, CT delivers the error logs and modification guidelines to the AI. Based on this, AI generates tests with the errors corrected.

In Case of Build Success, but Insufficient Coverage (PATH 2: BUILD SUCCESS & LOW COVERAGE)

If compilation succeeds but the coverage goal is not achieved, CT transmits information on uncovered branches and statements to the AI. AI then generates additional tests to execute those paths.

The loop terminates once the coverage goal is reached.

Performance

Quantitative Performance

The conventional AI test generation approach, as explained in Chapter 2, provides only code snippets and basic prompts. Confirmation of compilation success had to be done manually, and AI could not reflect project-level context or error feedback. The analysis-enriched approach enables AI to recognize project information and execution results, thereby improving test quality. It can increase both compilation success rates and coverage with minimal developer intervention.

The following results are based on experiments conducted on five open-source C++ projects (tinyclang, ninja, json, yaml, and re2 / total 49,012 LOC). C++ has a higher test design complexity than C, due to numerous complex language elements such as templates, class inheritance, namespaces, and operator overloading.

Basic Prompt Approach (Conventional AI Generation)

Table 3. Compilation Success Rate and Coverage of the Basic Prompt Approach

Project	LOC	Compilation Success Rate (CSR)	Coverage (Cov)
tinyclang	3,629	0%	0%
ninja	10,620	0%	0%
json	3,119	4.1%	27.5%
yaml	12,515	5.4%	16.5%
re2	19,129	0%	0%

Using only basic prompts, most projects fail to even compile.

CT Algorithm-Based Generation (Non-AI)

Table 4. CT's Algorithm-Based Generation (CSR & Coverage)

Project	Compilation Success Rate (CSR)	Coverage (Cov)
tinycl2	88.0%	58.6%
ninja	70.9%	29.3%
hjson	98.2%	59.4%
yaml	92.2%	49.3%
re2	93.4%	59.7%

CT's algorithm-based generation creates tests based on analysis information and according to rules. While it shows a stable compilation success rate, it has limitations in covering complex logic paths.

CT's Analysis-Enriched AI Test Generation

Table 5. Analysis-Enriched Generation (CSR & Coverage)

Project	LOC	Compilation Success Rate (CSR)	Coverage (Cov)	CT + AI Integrated
tinycl2	3,629	89.5%	91.4%	92.3%
ninja	10,620	92.6%	57.5%	61.3%
hjson	3,119	92.6%	90.0%	91.2%
yaml	12,515	91.9%	72.4%	81.8%
re2	19,129	85.3%	71.3%	76.8%
LOC Weighted Average	-	89.6%	71.3%	76.8%

Analysis-Enriched AI Generation achieved an average compilation success rate of 89.6% and 71.3% coverage. Merging the results of algorithm-based generation and Analysis-Enriched AI Generation raises the final coverage to 76.8%. This results from the two approaches complementing each other by covering different areas. Increasing coverage by an additional 5% from 70% is far more difficult than reaching 5% from 0%.

This coverage improvement is not merely a result of an increased number of tests. CT identifies unexecuted branches and conditional statements through Control Flow Graph (CFG) analysis, and transmits this information to the AI to induce the generation of additional tests. Based on this feedback, AI supplements missing execution paths and improves test quality.

Qualitative Performance

Minimized Developer Intervention

Test generation, compilation, and achieving the coverage goal proceed automatically. Developers or testers can manage the test cycle simply by reviewing the generated tests and execution results, which significantly reduces the time and effort spent on test creation.

Reduced Test Blind Spots

CT analyzes the Control Flow Graph (CFG) of uncovered areas to identify uncovered conditional statements and branches. AI supports this task, significantly reducing the time and cost required to achieve high coverage.

Increased Effectiveness in Large Codebases

As the codebase grows, it becomes difficult for humans to manually design and maintain tests. The analysis-enriched approach operates based on a comprehensive project analysis, enabling stable application even to large-scale legacy code with many files or complex dependencies. The larger the scale, the greater the effect of automation through AI.

Streamlined Test Maintenance

When code changes, compilation errors can occur in existing test code. The compilation error correction feature of the feedback-based improvement loop can be used independently. AI can automatically correct compilation errors in existing tests and also automatically generate test code for newly added functions. This feature helps reduce the effort for test maintenance.

4 CI/CD Integration of AI Test Generation

The Analysis-Enriched AI Test Generation, as described in Chapter 3, can be incorporated into Command-Line Interface (CLI) and CI/CD environments. The entire process, from test generation to compilation and coverage measurement, can be iteratively executed within the development pipeline.

CI/CD-Based Automation Process

CI/CD pipelines function as a platform for automatically executing the AI quality improvement loop. They automatically perform the entire process of test generation, execution, and correction.

The automation process consists of five steps:

- **Code Change Detection:** The CI/CD system automatically identifies changed functions or new source files.
- **Automated Existing Test Recovery (Self-Healing):** If compilation errors occur in existing tests due to code changes, AI automatically corrects them.
- **New Code Test Generation:** Test code is generated for newly added functions using the analysis-enriched approach.
- **Compilation and Coverage Analysis:** The generated tests are built and run to analyze coverage. If the coverage goal is not achieved, tests are supplemented through the feedback loop.
- **Result Confirmation:** Test generation results and coverage can be confirmed on the CI/CD dashboard.

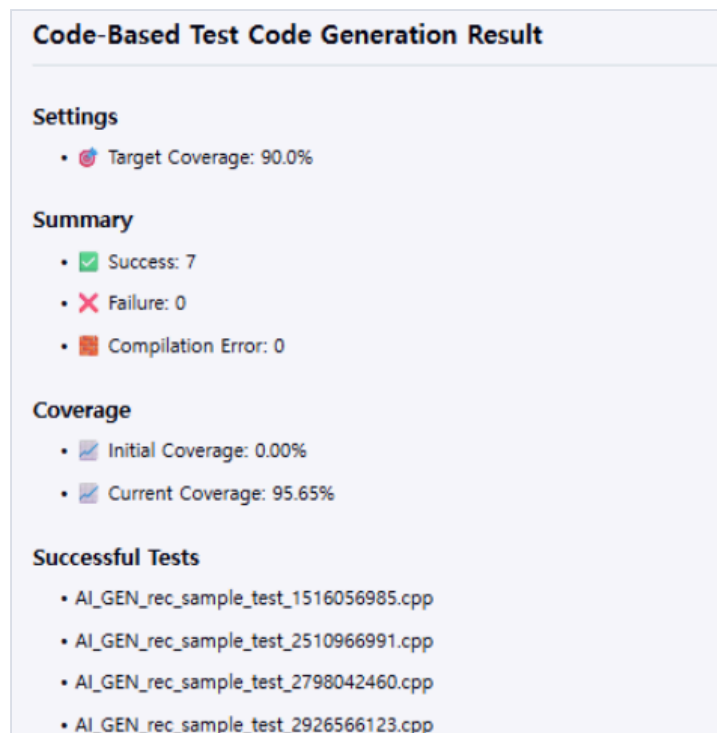


Figure 7 Code-Based Test Generation Results

Practical impact

Quality Improvement through Nightly Builds

Using high-performance LLMs can lead to longer processing times. By leveraging CI/CD nightly builds, even test generation requiring long computation times does not affect development schedules. Developers can arrive at work in the morning to find tests for code changed the previous day already generated.

Minimizing Test Gaps

When code changes, the CI/CD system automatically recognizes them and either automatically recovers existing tests or generates new ones in the next build. This reduces the risk of missed changes and ensures that tests for the latest code are always maintained.

5 Conclusion and Future Plans

CT's Analysis-Enriched AI Test Generation produces reliable tests even for C/C++ by providing accurate analysis information and leveraging a feedback loop. When incorporated into CI/CD environments, test generation, review, and correction automatically continue within the development pipeline, and tests become a quality asset that evolves with code changes.

Expected value

Reduced Time to Achieve Coverage Goals

Achieving high coverage requires designing tests that target difficult-to-reach branches and conditions. This task is time-consuming and demands experience. AI generates tests by identifying boundary values and exception paths that humans often overlook. This can significantly shorten the time required to achieve high-level coverage goals, such as MC/DC and branch coverage, which are demanded by functional safety standards.

Leveraging AI Test Generation in C/C++ Projects

C/C++ is one of the most challenging languages for AI test generation. Due to the abundance of information distributed outside the target source file, such as header dependencies, preprocessor macros, and type definitions, conventional AI tools struggle to even generate compilable tests. The analysis-enriched approach automatically collects and transmits this information to AI, thus allowing C/C++ projects to leverage the benefits of AI test generation.

Transforming Legacy Code into Test Assets

Adding tests to legacy code that has been operating without them is a burdensome task. This is because the codebase can be large, and maintainers may have changed. Since the analysis-enriched approach automatically analyzes the entire project, tests can be generated regardless of the maintainer's understanding of the code. This enables the transformation of legacy code into manageable test assets.

Future Plans

Currently, AI test generation focuses on improving compilation success rates and coverage. The semantic accuracy of tests, meaning whether the expected outcome is correct, must be reviewed by the user. Confirming that AI-generated tests operate as intended by the code is the responsibility of developers or testers.

CT plans to build a more precise quality automation system based on its analysis-enriched approach.

Requirement-Based Test Generation

Currently, AI test generation operates primarily around code structure and coverage. We are researching and developing a feature to generate tests that include expected outcomes, taking requirement documents or specifications as input. This aims to enhance the semantic accuracy of tests and reduce the user's review burden.

Improving AI's Judgment

We plan to improve AI's judgment capabilities to more accurately interpret quality metrics such as coverage, compilation results, and execution logs, and to more deeply understand the code's semantic structure and execution context. The goal is to maintain stable test generation quality regardless of project scale or complexity.

Token Optimization and Cost Reduction

CT's AI test generation feature supports local models. While local models face context window limitations, making it difficult to achieve results comparable to large cloud LLMs, we are working to reduce the number of tokens used in AI tasks and improve efficiency. This effort not only enhances the performance of local models but also effectively reduces costs when using cloud LLMs.

Adaptation to Latest Models

New LLM models are being released multiple times within just a few months. CT plans to continuously optimize prompts and tune results to match the characteristics of the latest models. We plan to respond quickly to ensure that test generation quality improves alongside model advancements.

CT's Analysis-Enriched AI Test Generation provides an environment where AI can generate reliable tests in C/C++ projects and automatically maintain and correct tests in response to code changes. Through this technology, developers can be freed from the burden of test creation, allowing them to focus on core tasks, and more efficiently manage the quality of mission-critical software.