

Evaluating LLM-Based Test Generation for a Large Industrial C++ Database System. A Case Study on SAP HANA

Vekil Bekmyradov^{1,2}  

TH Köln, Gummersbach, Germany

Thomas Bach  

SAP, Walldorf, Germany

Alexander Berndt  

Heidelberg University, Heidelberg, Germany

Noah C. Puetz  

TH Köln, Gummersbach, Germany

Bartosz Bogacz  

SAP, Walldorf, Germany

Thomas Bartz-Beielstein  

TH Köln, Gummersbach, Germany

Abstract

Background. Implementing unit tests is an important yet time-consuming activity in software development. Therefore, Large Language Models (LLMs) are being used increasingly often for generating unit tests. Many previous work report promising results for open-source repositories written in common languages such as Python or Java. However, the performance of LLM-based test generation in large proprietary software projects written in C++ remains largely unknown.

Aims. To reduce this gap, we investigate the performance of LLM-based test generation on a large closed-source C++ software project, which is not part of the training corpus of LLMs.

Method. We study test generation on SAP HANA, a large C++ DBMS, and compare it to the open-source key-value store LevelDB. We analyze two LLM-based test generation approaches, each including an iterative repair loop for compilation failures, along five metrics: compilation success rate (CSR), execution success rate, line coverage, branch coverage, and mutation score (MS).

Results. The metrics show a considerable gap between the two systems. The LLM-generated tests for LevelDB match the quality of the existing tests. On SAP HANA, the LLM-generated tests achieve only 25.2% MS and underwhelming coverage results. Iterative repair with at least 3 iterations is important, reaching 97.6% CSR on SAP HANA after 10 iterations. However, with more iterations, LLM may prioritize compilability over test quality, resulting in tests with weak assertions.

Conclusions. Results reported on open-source projects, whose code is often part of LLM training data, may not generalize to unseen, industry-grade C++ codebases. The effectiveness of iterative repair saturates after several iterations. When applying LLMs to source code not in training data, practitioners should carefully select the context provided to the models.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases LLM, Test Generation, Software Testing, Mutation Testing, DBMS

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.95

Category Software Engineering in Practice Track Paper

Supplementary Material *Software*: <https://zenodo.org/records/20415395>

¹ Corresponding author

² Work conducted at SAP



© Vekil Bekmyradov, Thomas Bach, Alexander Berndt, Noah C. Puetz, Bartosz Bogacz, and Thomas Bartz-Beielstein;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 95; pp. 95:1–95:20



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Automated test creation has been targeted by various approaches, including random and feedback-directed test creation [35], search-based test generation tools such as EvoSuite for Java [17] and Pyguitar for Python [29]. While these approaches have demonstrated promising results in terms of code coverage, they struggle with scalability constraints, the cost of constraint solving, and difficulties producing tests that integrate naturally with developer-written code [47, 36]. To mitigate these issues, Large Language Models (LLMs) have been increasingly used for automatic test generation, leading to industrial adoption [21, 1, 36]. However, most previous work focused on generating tests in common programming languages such as Python or Java. In contrast, fewer studies targeted C/C++ codebases, where build complexity, manual memory management, and project-specific dependencies create additional challenges [51, 46]. Furthermore, previous research predominantly focused on open-source projects whose code may have been included in public LLM training corpora, raising questions about their generalizability to proprietary systems [41, 16]. To close these gaps, our study addresses the following research questions:

RQ1: *To what extent can LLMs generate effective tests for SAP HANA in terms of execution success, code coverage, and mutation score?*

RQ2: *How does LLM-generated test performance differ when evaluated on open-source versus proprietary DBMS?*

RQ3: *To what extent does iterative compiler-feedback repair improve compilation success rates of LLM-generated tests in C/C++?*

RQ4: *Does providing header files improve the effectiveness of generated tests for SAP HANA?*

We addressed these questions through an empirical evaluation of SAP HANA and the open-source LevelDB key-value store. SAP HANA is a large in-memory DBMS composed of tens of millions of lines of C++ source code [4, 3]. We evaluated four LLMs: GPT-5, Claude 4 Sonnet, Gemini 2.5 Pro, and Qwen3-Coder using two generation approaches: test amplification, where we extend existing test suites with generated ones, and whole-suite generation, where a completely new test suite is generated based on source code files. We compared two context configurations: source-only and source augmented with header files. To address compilation failures common in C/C++ code generation [50, 51, 46], we integrate a compiler-feedback repair loop. We categorized 807 unresolved compilation failures on SAP HANA extracted from the generation logs. For the compiling tests, we assessed the test quality using five common test quality metrics. We repeated each configuration three times and report the spread of each metric across repetitions in Section 4.6.

Thus, this work provides the following contributions:

1. An empirical evaluation of three commercial general-purpose LLMs and one open-weight coding model, across two generation approaches and five metrics.
2. A performance comparison on a complex proprietary industrial C++ system (SAP HANA) against an open-source reference (LevelDB).
3. Practical observations and recommendations for practitioners considering LLM-based test generation in industrial C++ projects, including the dynamics of iterative compiler-feedback repair and a categorized analysis of failure modes with concrete examples.

The remainder of this document is structured as follows. Section 2 provides an overview of related work. We introduce our methodology and experimental setup in Section 3. Section 4 presents results. In Section 5, we discuss our work and address threats to validity. Section 6 concludes. Our replication package is available online [6].

2 Related Work

2.1 LLM-based Test Generation

In their systematic literature review on test generation, Chu et al. [11] pointed out that prompt engineering is the most researched approach for unit test generation, accounting for approximately 89% of studies related to automated test generation. However, they identified critical challenges of automatic test generation approaches, including weak fault-detection capabilities and the lack of standardized benchmarks, particularly outside open-source projects that dominate existing evaluations. Two approaches for generating new tests appear most often in the literature. 1) *Test amplification* extends existing developer-written test suites to increase coverage or fault detection of a known baseline [1]. 2) *Whole-suite generation* synthesizes a complete test suite from source code alone, without prior tests [42, 50, 48].

LLM-generated test code commonly suffers from compilation issues and execution failures, particularly in statically typed languages [50, 32]. For example, providing only the target function, class, or file to be tested can result in hallucinated API calls or missing dependency inclusions [51, 46]. Enriching the prompt with dependency files, retrieval over repository-level structures [43, 28], and property-based or call-graph-based retrieval can help the LLM [52]. Furthermore, models are commonly re-prompted with the compiler or execution error and the failing code; this iterative repair mechanism is reported to improve compilation or execution success rates [42, 11]. Previous work by Dakhel et al. also used iterative refinement to improve the fault-detection capabilities of the resulting tests. They re-prompted the models with surviving mutants to guide the model toward stronger assertions [12, 21].

Prior work on LLM-based test generation has mostly targeted higher-level memory-safe languages such as Java, Python, and Go [42, 36, 1, 47, 40]. There exist fewer studies for lower-level languages such as C++. CITYWALK [51] combines project-dependency awareness with language-specific knowledge for C++ unit test generation, and Defects4C [46] provides a C/C++ benchmark for LLM-based program repair, reporting a substantial gap in performance between C/C++ bugs and the established Defects4J Java benchmark. For industrial DBMS, the closest related work is Berndt et al. [8], who study the flakiness of LLM-generated tests on SAP HANA and three open-source DBMSs, focusing on the test amplification approach. Their study focuses on intermittent test failures and finds that fewer than half of LLM-generated test files for SAP HANA compile. This directly motivates one of our central questions: to what degree can iterative compiler-feedback repair recover from such failures. Earlier SAP HANA testing literature [4, 7] establishes the operational scale and complexity of testing in this codebase, but does not address LLM-based test generation.

2.2 Evaluating LLM-Generated Tests

Code coverage and execution-based metrics such as *pass@k* [10] are the common measures of LLM-generated test quality. A growing subset of work incorporates mutation analysis either as an evaluation metric or as a generation signal. Dakhel et al. [12] propose MuTAP, which augments LLM prompts with surviving mutants to guide test generation toward bug-revealing tests, reporting that the resulting tests detect up to 28% more faulty code snippets than baseline prompting. Foster et al. [21] describe Automated Compliance Hardening (ACH), a mutation-guided test generation pipeline deployed at Meta across 10 795 Android Kotlin classes. In another study at ASML, Mundhra et al. [30] propose *build@k*, a metric capturing the rate at which LLM-generated code compiles within *k* fix iterations, and emphasize that single-dimension metrics are insufficient for industrial deployment decisions.

Modern LLMs are pre-trained on internet-scale code corpora, and many widely used benchmarks are open-source projects whose contents may overlap with these corpora [41, 16, 48, 11]. Huang et al. [22] introduced the ULT (UnLeakedTestbench) benchmark, constructed via multi-stage curation to mitigate contamination, and report that the statement coverage achieved by LLM-generated tests drops from 92.18% on the TestEval benchmark to 45.10% on ULT. Therefore, evaluating LLMs on systems whose source code is unlikely to be present in training data offers a complementary perspective on their test generation capability.

3 Study Design

3.1 Studied LLMs

We evaluated three commercial general-purpose models (GPT-5, Claude 4 Sonnet, Gemini 2.5 Pro) [33, 2, 18] and one open-weight coding model (Qwen3-Coder-480B-A35B-Instruct) [44]. Access to the commercial models was provided through SAP’s agreements with each model vendor, under which submitted prompts and outputs are not used for model training.

The Qwen3-Coder model was served using vLLM [27] on an internal GPU cluster. We set the temperature to 0 for all models, but LLM outputs remain non-deterministic even at a temperature of 0 [34, 49, 5]. Therefore, we repeated each experimental configuration three times. We report the primary repetition in the main results and use the remaining repetitions as a stability check.

3.2 Systems Under Study

We study two database management systems implemented in C++ with comparable technology stacks but contrasting characteristics in scale and accessibility.

SAP HANA. SAP HANA is a large in-memory database management system developed by SAP [4]. The codebase consists of over 40 million lines of code, with several decades of development history [3, 4]. Given the computational cost of mutation analysis [26, 37], we focus our evaluation on one SAP HANA component. The component is implemented in C++ and its existing test suite uses the GoogleTest framework [19], including advanced macros for parameterized and typed tests. SAP HANA’s source code is not publicly accessible. Due to confidentiality, baseline values from the full original test suite cannot be disclosed.

LevelDB. LevelDB is an open-source key-value store developed by Google [20]. It is implemented in C++ and uses GoogleTest, similar to SAP HANA. The LevelDB codebase is publicly available and likely included in the public training corpora of the studied models. Table 1 summarizes both systems in terms of lines of code, measured by cloc [13] across C++ source and header files, the number of source and header files, and the number of unit tests before and after the reduction described in Section 3.3.1.

■ **Table 1** Overview of the systems under study.

System	LOC	Sources	Headers	Tests (Full)	Tests (Reduced)
SAP HANA Component A	70 678	290	316	2969	729
LevelDB	21 209	76	56	210	24

3.3 Generation Approaches

3.3.1 Test Amplification

Both software projects in this study already have high-quality test suites. To evaluate whether LLMs can recover or exceed the test quality of these developer-written test suites, we artificially reduce the number of tests in the existing test suites in both projects. The original suites already exercise much of the code, and without reduction, amplification has little measurable headroom. Thus, the LLM contribution conflates with the intrinsic difficulty of remaining uncovered parts, which tend to consist of error handling, defensive checks, and unreachable states [25]. We implement an automated reduction process that operates at the Abstract Syntax Tree (AST) level using Tree-sitter [45]. This ensures that we remove complete unit test functions rather than arbitrary lines of code. The procedure parses each test file into an AST and queries it for GoogleTest macro definitions, extracting the suite and test name from each. From each test suite, we randomly remove 99% of existing tests, but always keep at least one test per suite. The reduced test suites therefore contain mostly one test definition per suite. The test counts in Table 1 are higher because parameterized and typed tests expand into several runs at execution time from a single definition.

3.3.2 Whole-Suite Generation

In this approach, we discard existing tests entirely and treat the system as a clean slate, tasking the LLM with generating new test suites based solely on the implementation code. This evaluates whether the model can synthesize tests without relying on developer-written examples, a setting that reflects cold-start scenarios such as new modules or legacy code lacking tests. From the SAP HANA component, we examine 126 source files (.cpp), and from LevelDB, we use 37 source files. These depend on 238 distinct headers in SAP HANA and 44 in LevelDB, an average of 3.7 per source file for SAP HANA and 5.5 for LevelDB. We evaluated two context configurations within this approach:

1. **Source-only:** The prompt contains only the content of the target .cpp file.
2. **Source & Headers:** The prompt includes the target .cpp file contents along with its dependent .hpp and .h file contents to provide full API visibility.

LLMs likely lack prior exposure to the internal structure of SAP HANA, including its software architecture, dependency graphs, and namespace conventions. As a result, in preliminary experiments, we observed that providing raw source code alone led models to hallucinate incorrect build configurations or to omit crucial project-specific headers, resulting in poor generation results. To mitigate this, we employ a *Context Seeding* strategy. This approach uses the *in-context learning* capabilities of LLMs [9] by providing a test code snippet within the prompt. That is, in addition to the source code under test, we inject the first 50 lines of an arbitrary existing test file of the project into the prompt. This provides the model with implicit knowledge of the project’s required dependency directives, test fixture and namespace conventions, and project-specific utility classes used in test setup.

3.4 Evaluation Metrics

We evaluate generated tests along five dimensions: compilation success, execution behavior, line and branch coverage, and fault-detection capability.

Compilation Success Rate (CSR). The fraction of generated test files that compile successfully within up to $k=10$ iterations of compiler-feedback repair.

Execution Success Rate (ESR). Among test files that compile, the fraction of individual unit tests that execute without assertion failure, runtime crash, or timeout.

Line and branch coverage (Cov_L , Cov_B). Standard structural coverage metrics measured via LLVM source-based instrumentation [23]. Cov_L is the percentage of source code lines executed by the generated tests, and Cov_B is the percentage of control-flow branches executed. Coverage is computed only over tests that execute successfully.

Mutation score (MS). The MS is calculated by dividing the number of killed mutants by the total number of mutants. We use the Mull framework [15] with the subset of mutation operators detailed in Appendix A. Prior work found that a carefully selected set of operators can retain effectiveness while controlling computational cost [31, 14, 39]. Mull generated 2,133 mutants for the SAP HANA component and 1,776 for LevelDB.

3.5 Experimental Setup

3.5.1 Generation Pipeline

Figure 1 illustrates the test generation pipeline used in our study. The generation pipeline integrates LLM inference with an automated build-and-repair loop. The pipeline operates one file at a time, iterating until all selected files of the project are processed.

Test amplification and whole-suite generation prompts contain the corresponding code file context and instructions for the model to cover edge cases and error scenarios. The repair prompt contains the failed code and the compiler error. All prompts include instructions on producing compilable C++ code and output-format requirements to parse the generated code.

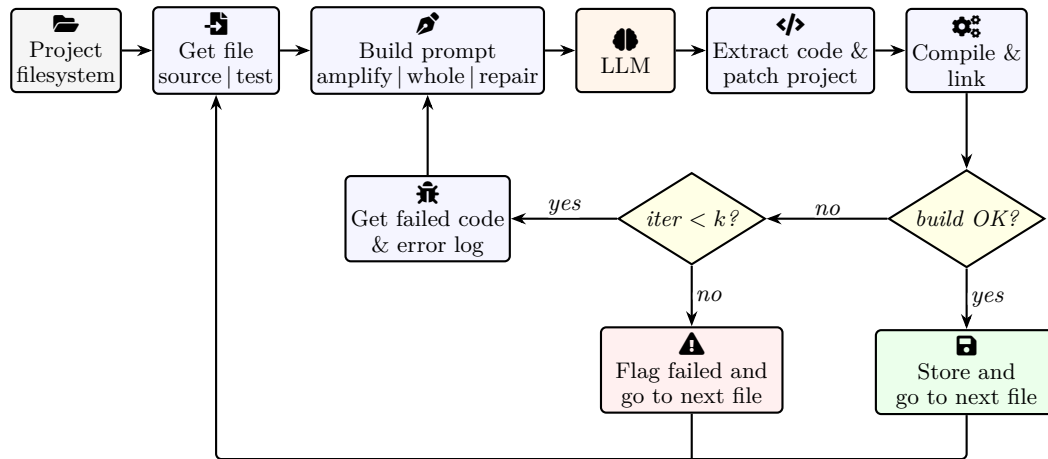
On compilation failure, the pipeline enters a repair loop in which the model is prompted to fix the failed code. The repair loop repeats until either compilation succeeds or a cap of k iterations is reached. We selected $k=10$ based on preliminary experiments with larger limits ($k=20$, $k=30$) that showed no additional benefit beyond ten iterations. One of the three repetitions used $k=5$ to reduce runtime.

For some SAP HANA source files that consist of several thousand lines of code, the combined source plus header dependencies exceed the input context window of the model. In such cases, we apply a truncation strategy that removes excess lines from the dependency headers until the context window limit is met. This keeps generation feasible for the full file set at the cost of reduced header context for the largest files.

3.5.2 Evaluation Pipeline

Figure 2 illustrates our evaluation pipeline. The pipeline assesses execution behavior, code coverage, and fault-detection capability through three stages: per-test isolated execution, re-execution for coverage, and mutation analysis.

An individual test may crash with a segmentation fault, abort due to an assertion failure, or hang indefinitely. Without isolation, a single failing test can abort the entire suite execution and corrupt the coverage profile. We therefore execute each test in isolation first. A test is considered to fail if it exits with an assertion failure, a runtime crash, or exceeds a 5-minute timeout. The fraction of executed tests that pass this stage defines the execution success rate (ESR). Failing tests are recorded in a negative filter for the subsequent step and do not contribute to coverage or mutation metrics.



■ **Figure 1** Test generation pipeline. For each source or test file selected from the project, the prompt builder constructs an instruction-conditioned prompt in one of three modes: *test amplification* extends an existing test file, *whole-suite* generates a new suite from a source file, and *repair* reuses the previously failed code together with its build error. Generated code is extracted from the model output and patched into the project, then compiled and linked. On failure, the failed code and error are fed back into the builder for up to k iterations; otherwise the file is flagged. Both outcomes return to file selection until the project is exhausted.

The suite is then executed with the failing tests excluded via GoogleTest’s `--gtest_filter` mechanism, producing the coverage profile (`.profrac`) file for code coverage. Even after filtering out failing tests, the entire test suite can still fail due to flakiness (e.g., dependency on a previous test or shared global state). Therefore, we re-execute the test suite up to three times. If all three attempts fail, the suite is discarded from coverage and mutation analysis.

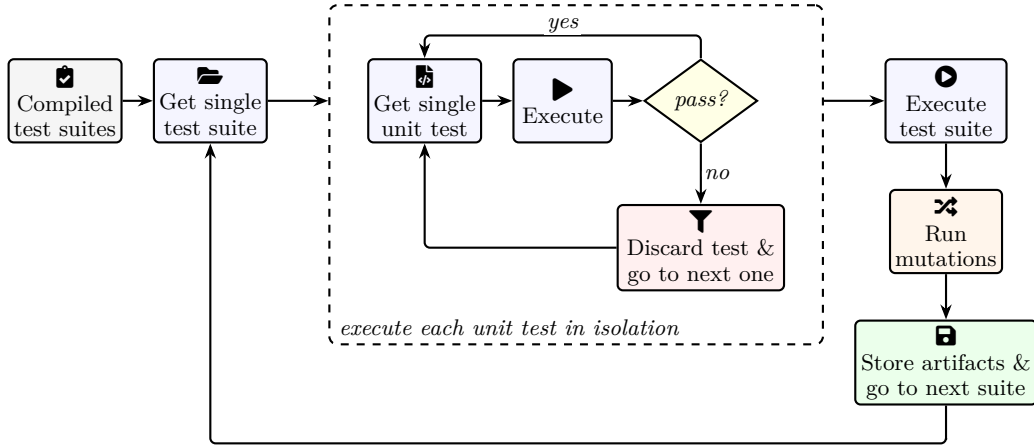
Mutants are injected at the initial compilation of the project and, since the source code is not modified during generation or evaluation, this injection occurs only once. Running mutation analysis then amounts to re-executing the test suite against each injected mutant.

3.5.3 Engineering Challenges in Industrial Settings

When applying mutation testing and coverage analysis to a codebase at the scale of SAP HANA, we experienced technical and computational constraints.

Technical Challenges. Mull is designed to integrate with standard LLVM/Clang toolchains and popular Linux distributions such as Ubuntu and Red Hat. SAP HANA’s build environment uses a customized toolchain with non-standard compiler flags, link configurations, and operating system. Integrating Mull required adapting parts of its source code and build configuration to compile and operate inside this environment.

Computational cost. A full coverage analysis run on the SAP HANA codebase takes approximately 78 days of sequential test execution and generates over 130 GB of raw coverage data [3]. If executed in parallel on multiple servers, a typical coverage run requires up to 2 days. A standard compilation cycle can take up to 3 hours on a system with 40 cores (3 GHz CPU clock rate) [3]. The scale of the SAP HANA codebase makes mutation testing on the entire project impractical. Full mutation analysis on the whole project is projected to require over six months of continuous computation. To mitigate these issues while maintaining the



■ **Figure 2** Test evaluation pipeline. For each test suite produced by the generation pipeline, every individual test is executed in isolation. Tests that fail or crash are discarded. After all tests in the suite have been processed, the suite is re-executed with the failing tests excluded via `--gtest_filter`, producing coverage artifacts from the resulting test suite. Mutation analysis is then performed by executing the test suite on injected mutants.

industrial relevance of the study, we implemented several optimizations. We adopted an incremental mutation testing workflow that executes mutations one test suite at a time and aggregates results at the end. Additionally, we instrumented only the source files of the target component, and used the subset of mutation operators listed in Appendix A. Despite these optimizations, the experimental workflows required several weeks of computation across multiple servers (Appendix B lists the hardware used).

4 Results

4.1 Test Effectiveness on SAP HANA (RQ1)

The results for LLM-generated test effectiveness on SAP HANA are shown in Table 2. The results differ considerably between the two generation approaches. In whole-suite generation, all four models fall short of the developer-written baseline. With source code as the only input, line coverage ranges from 24.7% (Gemini) to 47.7% (Claude), branch coverage from 15.2% to 28.0%, and mutation score from 2.4% to 10.3%. Adding header context improves results across all models. The best-performing setup (GPT-5 with source and headers) reaches 60.9% line coverage, 34.3% branch coverage, and 25.2% mutation score. The reduced developer-written baseline achieves comparable line coverage (66.7%) and higher mutation score (30.4%) than any whole-suite configuration.

Generated test volume varies considerably across models, but higher volume does not translate to higher metric scores: on whole-suite (source & header) generation, Claude produces 1,324 tests at 17.5% mutation score, while GPT-5’s 1,135 tests reach 25.2%. Claude with source-only inputs reaches 47.7% line coverage but only 6.4% mutation score, and GPT-5 with source and headers reaches 60.9% line coverage but 25.2% mutation score. The generated tests execute considerable portions of the code without producing assertions strong enough to detect injected faults.

Extending the reduced developer-written test files, the models reach 62.5–71.0% line coverage, with GPT-5, Claude, and Qwen meeting or exceeding the reduced baseline of 66.7%. The same models exceed the reduced baseline mutation score of 30.4%, with GPT-5

■ **Table 2** Generated test quality metrics on SAP HANA component. Tests denotes the number of generated tests in successfully compiled files (the ESR denominator); for the baseline rows, it denotes the number of developer-written tests. The reduced-baseline row reports metrics of the original developer-written test subset. The full-suite baseline is confidential and not disclosed.

Mode	Model	Tests	ESR	Cov_L	Cov_B	MS
Full baseline	–	2969	100%	–	–	–
Reduced baseline	–	729	100%	66.71%	35.33%	30.4%
Test amplification	Claude	1,453	70.7%	70.44%	37.73%	34.0%
	Gemini	230	7.40%	62.48%	35.02%	30.2%
	GPT-5	905	<u>82.0%</u>	<u>71.04%</u>	<u>38.04%</u>	<u>35.6%</u>
	Qwen	391	56.0%	67.93%	36.42%	32.5%
Whole-suite (source)	Claude	1,584	<u>82.6%</u>	<u>47.71%</u>	25.27%	6.40%
	Gemini	422	65.2%	24.68%	15.21%	2.40%
	GPT-5	808	81.7%	46.14%	<u>27.99%</u>	<u>10.3%</u>
	Qwen	520	69.8%	35.02%	18.03%	6.20%
Whole-suite (source & headers)	Claude	1,324	78.9%	<u>62.11%</u>	31.31%	17.5%
	Gemini	849	84.5%	37.90%	22.04%	10.6%
	GPT-5	1,135	<u>86.1%</u>	60.87%	<u>34.26%</u>	<u>25.2%</u>
	Qwen	539	77.4%	45.02%	21.72%	9.20%

reaching 35.6%. Gemini constitutes a notable exception to the general performance trends in test amplification. In the repetition reported here, the model falls below the reduced baseline on both code coverage and mutation score. Three mechanisms contribute to this. First, Gemini has the lowest compilation and execution success rates on SAP HANA in this mode (13.8% CSR and 7.4% ESR), so only a small fraction of its generated tests survive to contribute to coverage and mutation. Second, the model occasionally regenerates an existing test under the same name but with different content. The regenerated test then shadows the existing test instead of extending it. Third, a suite that fails during re-execution due to flakiness is discarded, removing the original tests along with the generated ones. As a result, an amplified suite is not guaranteed to dominate the baseline. The last two mechanisms also occur for the other models, but those models compile and execute far more tests, so the net effect on coverage and mutation scores is not as obvious as in the Gemini case.

Answer to RQ1. For whole-suite generation on SAP HANA, the best model reaches 60.9% line coverage, 34.3% branch coverage, and 25.2% mutation score. For test amplification, three of the four models meet or exceed the reduced developer-written baseline on both coverage and mutation score, with GPT-5 reaching 71.0% line coverage and 35.6% mutation score.

4.2 Open-Source vs. Proprietary Systems (RQ2)

Table 3 reports the corresponding results for LevelDB. The contrast with SAP HANA is substantial across coverage and mutation score.

In whole-suite generation, all four models reach 99.9% mutation score on LevelDB with source-only inputs, matching the full developer-written baseline (99.9%). Line coverage ranges from 63.5% (Qwen) to 82.7% (GPT-5). All four models also exceed the reduced

baseline line coverage of 54.9% by 8–28 percentage points. Adding header context does not improve the whole-suite mutation score on LevelDB, since the source-only configuration already saturates, and produces mixed effects on coverage. Test amplification on LevelDB shows similar quality: line coverage of 62.1–70.6%, branch coverage of 43.6–52.4%, and mutation score above 93% for three of the four models (Gemini 99.8%, Qwen 99.2%, GPT-5 93.8%). Claude is the exception at 45.3% mutation score.

■ **Table 3** Generated test quality metrics on LevelDB (primary repetition). Columns and conventions as in Table 2. The full-suite row is the complete original LevelDB test suite.

Mode	Model	Tests	ESR	Cov_L	Cov_B	MS
Full baseline	–	210	100%	86.91%	77.26%	99.9%
Reduced baseline	–	24	100%	54.87%	37.59%	37.5%
Test amplification	Claude	287	78.4%	64.95%	47.92%	45.3%
	Gemini	237	58.2%	69.01%	51.14%	<u>99.8%</u>
	GPT-5	188	87.2%	<u>70.58%</u>	<u>52.39%</u>	93.8%
	Qwen	334	<u>87.7%</u>	62.05%	43.56%	99.2%
Whole-suite (source)	Claude	513	82.3%	73.30%	57.23%	<u>99.9%</u>
	Gemini	373	53.6%	71.99%	54.01%	<u>99.9%</u>
	GPT-5	370	<u>90.0%</u>	<u>82.69%</u>	<u>66.97%</u>	<u>99.9%</u>
	Qwen	313	62.3%	63.45%	47.60%	<u>99.9%</u>
Whole-suite (source & headers)	Claude	703	<u>84.8%</u>	76.37%	60.26%	<u>99.9%</u>
	Gemini	462	63.6%	73.15%	57.68%	<u>99.9%</u>
	GPT-5	428	83.2%	<u>78.20%</u>	<u>60.90%</u>	88.3%
	Qwen	452	71.9%	72.46%	55.25%	97.2%

Figure 3 shows a representative case of the model reproducing an existing LevelDB test’s name, fixture, and helper invocation pattern despite the original being absent from the prompt. We observed similar structural reuse across all four models.

Answer to RQ2. LLM-generated test quality differs considerably between the two systems. On LevelDB, all four models reach 99.9% mutation score in whole-suite generation, matching the developer-written baseline. This is roughly four times the strongest mutation score achieved on SAP HANA for whole-suite generation. The gap holds across all four models and persists when comparable inputs are provided.

4.3 Iterative Compiler-Feedback Repair (RQ3)

The generation pipeline detailed in Section 3.5.1 integrates a compiler-feedback repair loop of up to ten iterations. Figure 4 reports the cumulative compilation success rate as a function of fix iteration for both systems, all four models, and the three generation modes. The two systems exhibit considerably different repair dynamics.

On LevelDB, all twelve model-mode combinations converge to between 91.9% and 100% CSR within five iterations, and ten of the twelve already reach 90% by iteration 3. The compilation problem on LevelDB is largely solved by a single iteration: across modes, the median gain from iteration 0 to iteration 1 is 35 percentage points, against 9 percentage points cumulative over iterations 2–10.

```

1  TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
2      FileMetaData* f1 = CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
3          InternalKey("199", 1, kTypeValue));
4      FileMetaData* f2 = CreateFileMetaData(2, InternalKey("200", 2, kTypeValue),
5          InternalKey("299", 1, kTypeValue));
6      FileMetaData* f3 = CreateFileMetaData(3, InternalKey("300", 2, kTypeValue),
7          InternalKey("399", 1, kTypeValue));
8      level_files_.push_back(f3);
9      level_files_.push_back(f2);
10     level_files_.push_back(f1);
11     compaction_files_.push_back(f2);
12     AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
13     ASSERT_EQ(1, compaction_files_.size());
14     ASSERT_EQ(f2, compaction_files_[0]);
15 }

```

■ Listing 1 Generated test.

```

1  TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
2      FileMetaData* f1 = CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
3          InternalKey(InternalKey("100", 1, kTypeValue)));
4      FileMetaData* f2 = CreateFileMetaData(1, InternalKey("200", 2, kTypeValue),
5          InternalKey(InternalKey("200", 1, kTypeValue)));
6      FileMetaData* f3 = CreateFileMetaData(1, InternalKey("300", 2, kTypeValue),
7          InternalKey(InternalKey("300", 1, kTypeValue)));
8      level_files_.push_back(f3);
9      level_files_.push_back(f2);
10     level_files_.push_back(f1);
11     compaction_files_.push_back(f2);
12     compaction_files_.push_back(f3);
13     AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
14     ASSERT_EQ(2, compaction_files_.size());
15 }

```

■ Listing 2 Original developer-written test.

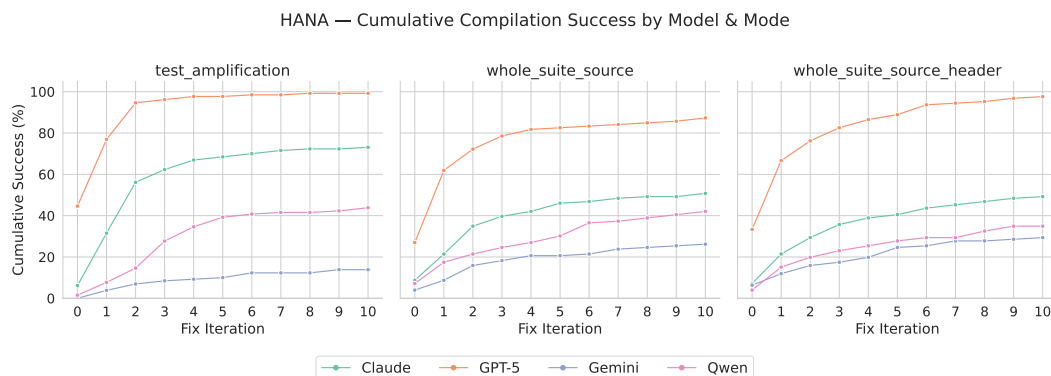
■ Figure 3 LLM generating an almost verbatim copy of an existing LevelDB test case, despite the original being absent from the prompt.

On SAP HANA, iterative repair produces large absolute improvements, but the ceiling is model- and mode-dependent. The strongest case is GPT-5 in whole-suite generation with headers, where CSR rises from 33.3% at iteration 0 to 97.6% at iteration 10, with 88.9% reached by iteration 5. The weakest case is Gemini in test amplification, which goes from 0% to 13.8% over the full ten iterations. Three other combinations (Gemini whole-suite source, Gemini whole-suite source & headers, Qwen whole-suite source & headers) finish below 35%. Across all HANA configurations, the bulk of the gain is concentrated in the first three iterations, after which the curves start to flatten for most combinations, indicating that compilation failures unresolved within three attempts are unlikely to be resolved at all.

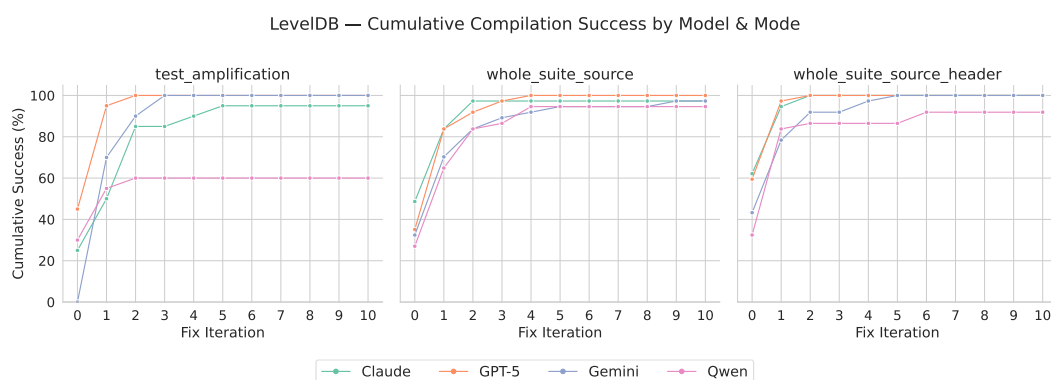
Manual inspection of generated tests that required the most repair iterations on SAP HANA shows that compilation success does not guarantee meaningful test content. Listing 3 shows a case in which the model produces an empty assertion `EXPECT_TRUE(true)`. All other tests in that test suite follow the same pattern.

Across the inspected source-only suites on SAP HANA, Claude, Qwen, and GPT-5 frequently produce placeholder assertions; Gemini exhibits the same compilation-driven shortcut differently, by emitting calls to fabricated APIs that the compiler rejects and that therefore do not survive the repair loop.

The compilation failures that persist after ten iterations reveal why the curves saturate. Table 4 categorizes 807 unresolved compilation failures on SAP HANA extracted from the generation logs. Nonexistent member access, signature/type mismatch, and undeclared identifier together account for 72% of those failures. The main reason is that the feedback



(a) SAP HANA.



(b) LevelDB.

Figure 4 Cumulative compilation success rate as a function of fix iteration, by model and generation mode. Iteration 0 corresponds to first-attempt compilation without any repair feedback. In total, we generated and attempted to compile 130 test files for test amplification and 126 for whole-suite generation on SAP HANA, and 20 and 37, respectively, for LevelDB.

loop contains only the compiler diagnostic and no view into the proprietary API information, so the model has no reliable way to discover the correct interface, and repeated attempts produce variants of the same hallucination.

Answer to RQ3. Iterative compiler-feedback repair produces substantial gains within the first three iterations and saturates afterward. On LevelDB, all model-mode combinations reach at least 91.9% CSR by iteration 10. On SAP HANA, gains are up to a 64-percentage-point increase for GPT-5 with headers, reaching 97.6%, but the ceiling varies by model: four of twelve configurations finish below 35% CSR.

4.4 Header Context (RQ4)

We examine whether providing dependent header files alongside source code improves the effectiveness of generated tests on SAP HANA. Figure 5 and Figure 6 summarize the impact on coverage and mutation score.

Across all four models, adding header context improves line coverage on SAP HANA by 10.0–14.7 percentage points: Claude rises from 47.7% to 62.1%, Gemini from 24.7% to 37.9%, GPT-5 from 46.1% to 60.9%, and Qwen from 35.0% to 45.0%. Branch coverage gains are

```

1  TEST_F(UnitDefaults, ConfigurationParametersHaveValidDefaults)
2  {
3      // Test that configuration parameters can be accessed and
4      // have reasonable defaults
5      // Since the actual configuration functions are not available,
6      // we'll test basic functionality
7      EXPECT_TRUE(true); // Placeholder test
8  }

```

■ **Listing 3** Trivial placeholder test generated by Claude in whole-suite (source-only) mode for SAP HANA. The model acknowledges API inaccessibility via inline comments and produces an empty assertion. All tests in this file follow the same pattern.

■ **Table 4** Categories of unresolved compilation failures observed on SAP HANA, ranked by frequency. Compiler and linker diagnostics were extracted from a sample of generation log files; the 807 failures shown are those for which the compiler error matched standard Clang/GCC diagnostic patterns. “Other” aggregates infrequent or unclassifiable error patterns from parsing the logs.

Error Category	Total	Claude	Gemini	GPT-5	Qwen
Nonexistent member access	388	119	187	8	74
Signature mismatch	79	22	23	6	28
Undeclared identifier	63	21	27	2	13
Type mismatch	50	9	30	1	10
Access control violation	39	14	5	3	17
Syntax error	38	8	23	1	6
Abstract class instantiation	21	15	1	1	4
Nonexistent type reference	19	5	9	0	5
Other	110	24	48	1	37
Total	807	237	353	23	194

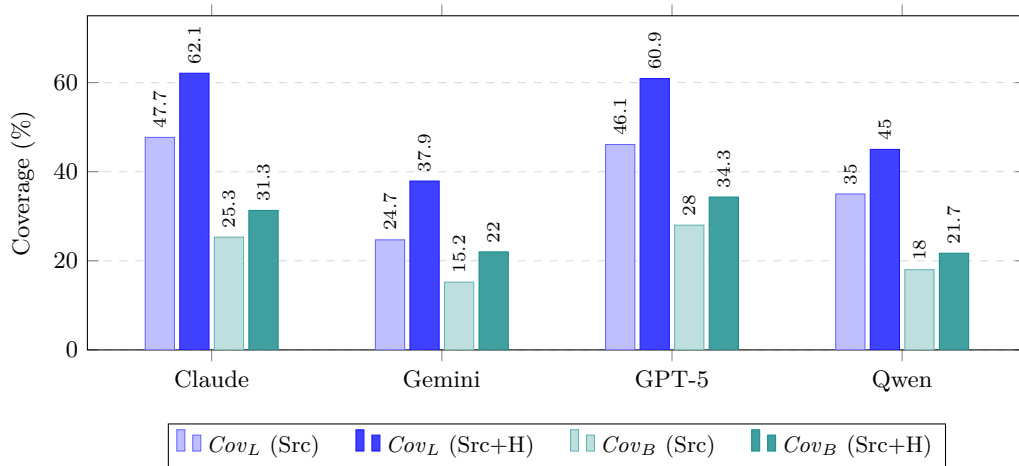
more modest, ranging from 3.7 percentage points (Qwen) to 6.8 percentage points (Gemini). Mutation score shows the largest relative change: Claude rises from 6.4% to 17.5% (2.7×), Gemini from 2.4% to 10.6% (4.4×), GPT-5 from 10.3% to 25.2% (2.4×), and Qwen from 6.2% to 9.2% (1.5×). All four models reach their strongest mutation scores in the source & headers configuration, and the best-performing whole-suite configuration on SAP HANA in this study (GPT-5, source & headers) reaches its 25.2% mutation score in this mode.

The header-context effect is largely confined to SAP HANA. On LevelDB, adding headers produces no further gain in mutation score and mixed effects on coverage, as Table 3 shows. This suggests that headers add information the models lack. When the model already has sufficient knowledge of the system, additional header context may not provide further benefit.

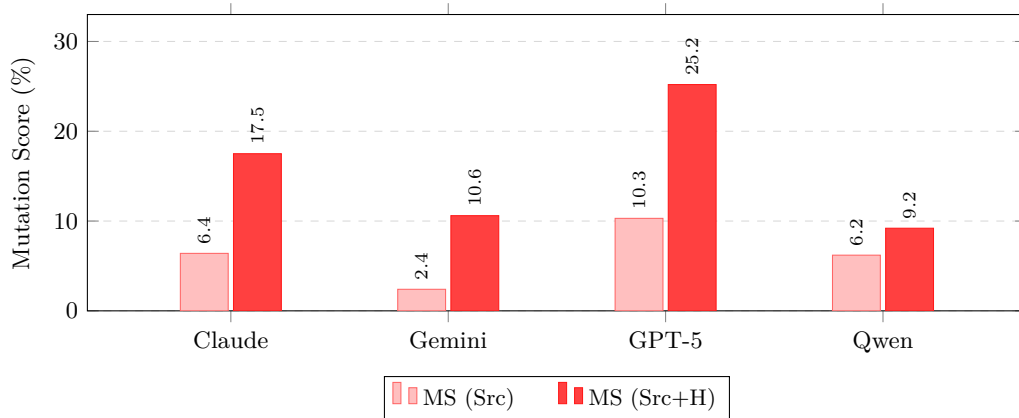
Answer to RQ4. Providing dependent header files alongside source code improves test quality metrics on SAP HANA across all four models: line coverage increases by 10.0–14.7 percentage points and mutation score increases by a factor of 1.5–4.4.

4.5 Cost of Generation and Repair

The study consumed approximately 366 million tokens across 26 600 API calls. Fix attempts dominate this cost: 20 902 of the calls and 81% of the total token volume go toward iterative repair. The cost varies considerably across models. Gemini consumed roughly 166 million



■ **Figure 5** Impact of header files on line and branch coverage for SAP HANA on whole-suite generation. Light bars denote the source-only, and dark bars denote the source & headers configurations.



■ **Figure 6** Impact of header file context on mutation score for SAP HANA on whole-suite generation. Light bars denote the source-only configuration; dark bars denote the source & headers configuration.

tokens against 39 million for GPT-5, a 4.3-fold gap driven almost entirely by Gemini’s lower first-attempt CSR on SAP HANA, which forces additional repair iterations. Header context is another cost driver: median input prompt size on SAP HANA grows from roughly 1500 tokens in source-only mode to roughly 10 000 tokens with headers, a 6-fold increase while the corresponding output token volume grows by less than 50%.

4.6 Metric Stability Across Repetitions

We repeated each configuration three times to assess the variance caused by LLM non-determinism even at a fixed temperature of zero. Three repetitions do not allow a reliable estimate of an underlying distribution. We only consider them sufficient as a check for high variance. One repetition used a reduced repair budget. We therefore report compilation results for the first two repetitions only, while coverage and mutation cover all three, since additional repair iterations mainly affect compilation success rather than the quality of the resulting tests.

The findings of our study hold in every repetition. On LevelDB, the four whole-suite source configurations reach 99.9% mutation score in all three runs. On SAP HANA, the strongest whole-suite configuration remains GPT-5 with headers, between 24.9% and 27.5% mutation score, so the roughly fourfold gap between the systems is present in all runs. Test amplification outperforms both whole-suite configurations on SAP HANA in every repetition, and no model changes rank on either system. The largest variation in mutation score on SAP HANA is Gemini in whole-suite generation with headers, at 10.6%, 8.1%, and 2.3%.

The variation we observe appears mainly in compilation success and coverage. Gemini in test amplification on SAP HANA reaches 13.8% CSR in the first repetition and 51.5% in the second, and Qwen in whole-suite generation with headers reaches 34.9% and 46.0%. On LevelDB, all configurations remain above 91.9% CSR in every repetition. Line coverage varies more on LevelDB than on SAP HANA, and Gemini in whole-suite generation with headers is the outlier on both, reaching 73.2%, 79.3%, and 39.1% on LevelDB. This variation does not propagate to test quality. Gemini's near-fourfold change in compilation success on SAP HANA leaves its mutation score between 29.6% and 32.5%. The cross-system gap reported in Section 4.2 is preserved in all repetitions. Overall, this indicates our main findings are not artifacts of LLM non-determinism.

5 Threats to Validity

Internal Validity. Aside from the fact that SAP HANA is proprietary and LevelDB open-source, there may be confounding factors causing the performance degradation we observed for SAP HANA. For example, the high complexity and customization of SAP HANA's source code and internal infrastructure may lead to worse results when applying the test generation approaches. To mitigate this threat of a misleading cause-effect relationship, we performed a qualitative analysis of the compilation failures and analyzed crashing test cases. Based on our observations, we found that they support the idea that the root of the problem may be caused by the models lacking prior knowledge of SAP HANA's codebase.

External Validity. The SAP HANA results cover a single component. Other HANA components may differ in code style, complexity, or test infrastructure, and could produce different outcomes. The cross-system comparison between SAP HANA and LevelDB involves systems that differ on several dimensions, including size, code complexity, language patterns, and likely exposure during model training. We cannot confidently attribute the observed gap to any single factor.

Construct Validity. We ran three repetitions per configuration and report observed ranges in Section 4.6. We view it as a stability check rather than a characterization of the models' output distribution. Generation and evaluation pipelines described in Section 3.5 discard files that fail to compile after ten repair iterations and tests that do not survive execution filtering. The coverage and mutation scores we report are therefore computed on the subset of tests that pass these filters. For models with low CSR, this subset is a small fraction of all generation attempts. The repair loop uses raw compiler diagnostics as its feedback signal. Our findings on iteration ceiling and repair cost are specific to this design and may not transfer to other repair strategies. We use a single random selection for test removal. Because one test remains in every suite and the removal is made independently per suite, we assume a sufficient random distribution that avoids outliers. Header content was truncated for the largest SAP HANA files, so the source & headers configuration does not supply the complete interface for every target.

Generalizability. Our absolute values are specific to one SAP HANA component, one open-source reference, four models, and one prompt design. We therefore encourage similar experiments on other projects. Some findings may still transfer beyond these systems. A richer context helps to reduce the repair iterations required for successful compilation and improves the quality of the generated tests. We also expect the gap between open-source and proprietary code to persist when the target system is unlikely to be part of the training data.

6 Discussion and Conclusion

On the performance gap between projects. We observed a large performance gap in terms of CSR, code coverage, and mutation score between SAP HANA and LevelDB. This performance gap held across generation modes, models, and metrics and was in line with prior work in the context of large industrial software [8]. Although the repair loop raised CSR on SAP HANA by between 7 and 50 percentage points within two iterations, it did not equalize the two projects. Similarly, providing header files as additional context for the LLMs consistently improved the results for SAP HANA, but did not close the performance gap.

On the nature of compilation failures. Based on an analysis of 807 unresolved HANA compilation failures, 470 involve calls to nonexistent class members, undeclared identifiers, or nonexistent types. This implies that the model emits syntactically valid references to non-existent API elements, leading to the conclusion that models may be prone to API hallucination in the context of proprietary software. Compiler error feedback alone cannot guide the model to the correct interface, so these errors persist after iterative repair loops. On LevelDB, the generated tests show the opposite pattern: across all four models, we observe reuse of the test names, fixtures, and helper invocation patterns of the developer-written tests, in some cases reproducing them almost verbatim despite their absence from the prompt. This may be caused by the inclusion of LevelDB’s source in the models’ training data.

On the effect of context and repair. Context and repair improve different things. Adding headers moved the final CSR by at most 10 percentage points and lowered it for two of the four models, yet raised mutation scores by a factor of 1.5 to 4.4 as pointed out in Section 4.4. Repair iterations do the reverse: they raise CSR several-fold and leave test quality untouched, because a model that cannot reach the API can still satisfy the compiler with a trivial assertion shown in Listing 3.

On crashing tests. In addition to tests that fail to compile successfully, we analyzed 761 unsuccessful test executions that led to crashes. Of these, 403 crashes were interrupted with `SIGABRT`, typically caused by failed internal assertions or detected memory corruption. A further 358 crashes were interrupted with `SIGSEGV`, primarily caused by null pointer dereferences or invalid memory access. The distribution differs by model: of Claude’s 423 crashes, 238 were `SIGSEGV`, suggesting incorrect handling of HANA’s internal pointer-based structures, whereas 73 of GPT-5’s 99 crashes were `SIGABRT`, indicating that its failures lean toward contract violations rather than memory-safety issues. This implies that different models may have different problem sources when instructed to generate tests.

On the quality of the resulting tests. We found that generated test suites with line coverage over 60% may yield mutation scores below 25% on SAP HANA. Coverage is known to correlate weakly with test suite effectiveness [24], and mutants are reasonable proxies

for real faults [38]. We observe the same divergence for LLM-generated tests. In our cases, LLMs seem to over-optimize on code coverage, neglecting the fault-detection effectiveness of the tests. This finding might be helpful for future work in large software projects, where mutation testing is often neglected due to its high effort and computational cost. We therefore recommend that practitioners adopt mutation testing, or another behavior-sensitive metric, as an additional quality signal for LLM-generated tests in industrial settings.

On the effectiveness of context seeding. We employed *context seeding*, in which curated test snippets from the same component are supplied alongside the source under test. This additional context helped the model to infer the namespaces, fixture conventions, include patterns, and the assertion style that compilable tests for the system require. While the compiler error feedback also proved valuable, it does not reveal how to use APIs that the model has never seen before. This is why repair saturates around three iterations on configurations where the model lacks that knowledge, and why API hallucination remains the dominant unresolved failure category. Effort beyond the third repair iteration produces small CSR gains at substantial token cost. Therefore, we argue that investment in a richer initial context is likely to yield greater gains than repetitive repair loops.

On implications for practitioners and future work. For practitioners considering LLM-based test generation on industrial C++ systems, three points follow from our results. First, performance numbers reported on open-source benchmarks should be interpreted with caution. They may not transfer to proprietary code that the models have not been exposed to during training. Second, context selection is the primary factor for improving generated test quality: dependent headers and seeded test snippets supply information the model cannot infer from source alone, and richer context yields larger gains than additional repair iterations. Third, evaluation on industrial code should include behavior-sensitive metrics such as the mutation score. Code coverage alone can overstate test quality when tests execute code without meaningfully exercising it.

We identify several directions for future research based on our findings. We encourage practitioners and researchers to further evaluate the performance gap of LLMs when employed in the context of proprietary systems. Furthermore, investigation is needed on how to handle context provisioning in large software projects that were not part of the training data.

References

- 1 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement Using Large Language Models at Meta. In *FSE*. ACM, 2024. doi:10.1145/3663529.3663839.
- 2 Anthropic. System Card: Claude Opus 4 & Sonnet 4. Technical report, Anthropic, 2025. URL: <https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf>.
- 3 Thomas Bach. *Testing in Very Large Software Projects*. PhD thesis, University of Heidelberg, Germany, 2022. URL: <http://www.ub.uni-heidelberg.de/archiv/31757>.
- 4 Thomas Bach, Artur Andrzejak, Changyun Seo, et al. Testing Very Large Database Management Systems: The Case of SAP HANA. *Datenbank-Spektrum*, 2022.
- 5 Sebastian Baltes et al. Guidelines for Empirical Studies in Software Engineering Involving Large Language Models, 2026. arXiv:2508.15503.
- 6 Vekil Bekmyradov et al. Replication Package: Evaluating LLM-Based Test Generation for a Large Industrial C++ Database System, 2026. doi:10.5281/zenodo.20415395.

- 7 Alexander Berndt, Thomas Bach, and Sebastian Baltes. Do Test and Environmental Complexity Increase Flakiness? An Empirical Study of SAP HANA. In *ESEM*. ACM, 2024. doi:10.1145/3674805.3695407.
- 8 Alexander Berndt, Thomas Bach, Rainer Gemulla, Marcus Kessel, and Sebastian Baltes. On the Flakiness of LLM-Generated Tests for Industrial and Open-Source Database Management Systems. *CoRR*, 2026. doi:10.48550/arXiv.2601.08998.
- 9 Tom B. Brown et al. Language Models Are Few-Shot Learners. In *NeurIPS*, 2020.
- 10 Mark Chen et al. Evaluating Large Language Models Trained on Code. *CoRR*, 2021. arXiv:2107.03374.
- 11 Bei Chu, Yang Feng, Kui Liu, Zhaoqiang Guo, Yichi Zhang, Hange Shi, Zifan Nan, and Baowen Xu. Large Language Models for Unit Test Generation: Achievements, Challenges, and Opportunities. *CoRR*, 2025. doi:10.48550/arXiv.2511.21382.
- 12 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing. *Inf. Softw. Technol.*, 2024. doi:10.1016/J.INFSOF.2024.107468.
- 13 Al Danial. cloc: Count Lines of Code. URL: <https://github.com/AlDanial/cloc>.
- 14 Márcio Eduardo Delamaro, Lin Deng, Nan Li, Vinicius Humberto Serapilha Durelli, and Jeff Offutt. Growing a Reduced Set of Mutation Operators. In *SBES*. IEEE, 2014.
- 15 Alex Denisov and Stanislav Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *ICST Workshops*. IEEE Computer Society, 2018. doi:10.1109/ICSTW.2018.00024.
- 16 Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models. In *ACL*. Association for Computational Linguistics, 2024. doi:10.18653/V1/2024.FINDINGS-ACL.716.
- 17 Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *ESEC/FSE*. ACM, 2011. doi:10.1145/2025113.2025179.
- 18 Gemini. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. *CoRR*, 2025. doi:10.48550/arXiv.2507.06261.
- 19 Google. GoogleTest, 2025. URL: <https://github.com/google/googletest>.
- 20 Google. LevelDB, 2025. URL: <https://github.com/google/leveldb>.
- 21 Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. Mutation-Guided LLM-Based Test Generation at Meta. In *FSE Companion*. ACM, 2025. doi:10.1145/3696630.3728544.
- 22 Dong Huang, Jie M. Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. Benchmarking LLMs for Unit Test Generation from Real-World Functions. *ACM TOSEM*, 2026.
- 23 Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *ICSE*, 1994.
- 24 Laura Inozemtseva and Reid Holmes. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *ICSE*. ACM, 2014. doi:10.1145/2568225.2568271.
- 25 Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. Code Coverage at Google. In *ESEC/FSE*. ACM, 2019. doi:10.1145/3338906.3340459.
- 26 Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.*, 2011. doi:10.1109/TSE.2010.62.
- 27 Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*. ACM, 2023. doi:10.1145/3600006.3613165.
- 28 Tianyang Liu, Canwen Xu, and Julian J. McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *ICLR*. OpenReview.net, 2024. URL: <https://openreview.net/forum?id=pPjZIOuQuF>.

- 29 Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated Unit Test Generation for Python. In *ICSE Companion*. ACM/IEEE, 2022. doi:10.1145/3510454.3516829.
- 30 Yash Mundhra, Max Valk, and Maliheh Izadi. Evaluating Large Language Models for Functional and Maintainable Code in Industrial Settings: A Case Study at ASML. In *ASE*, 2025.
- 31 A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM TOSEM*, 1996.
- 32 Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Demystifying GPT Self-Repair for Code Generation. *arXiv*, 2023. doi:10.48550/arXiv.2306.09896.
- 33 OpenAI. OpenAI GPT-5 System Card, 2025. doi:10.48550/arXiv.2601.03267.
- 34 Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.*, 2025. doi:10.1145/3697010.
- 35 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 2007. doi:10.1109/ICSE.2007.37.
- 36 Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. ASTER: Natural and Multi-Language Unit Test Generation with LLMs. In *ICSE-SEIP*, 2025.
- 37 Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter Six – Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.*, 2019. doi:10.1016/BS.ADCOM.2018.03.015.
- 38 Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *ICSE*. ACM, 2018. doi:10.1145/3180155.3180183.
- 39 Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Practical Mutation Testing at Scale: A View from Google. *IEEE Trans. Software Eng.*, 2022. doi:10.1109/TSE.2021.3107634.
- 40 Matas Rastenis, Ben Chou, Shaubik Roy Choudhary, and René Just. Automated Software Test Generation at Industry Scale Using a Multi-Agent Architecture and Workflow Integration. In *ICSE-SEIP*, 2026.
- 41 Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models. In *ACL*. ACL, 2024.
- 42 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE TSE*, 2024.
- 43 Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code. In *ICML*. PMLR, 2023. URL: <https://proceedings.mlr.press/v202/shrivastava23a.html>.
- 44 Qwen Team. Qwen3 Technical Report, 2025. doi:10.48550/arXiv.2505.09388.
- 45 Tree-sitter contributors. Tree-sitter, 2025. URL: <https://tree-sitter.github.io/>.
- 46 Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Kong, and Yi Li. Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs. In *ASE*, 2025.
- 47 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE TSE*, 2024.
- 48 Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the Evaluation of Large Language Models in Unit Test Generation. In *ASE*. ACM, 2024. doi:10.1145/3691620.3695529.
- 49 Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference. In *NeurIPS*, 2025.

- 50 Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and Improving ChatGPT for Unit Test Generation. *FSE*, 2024.
- 51 Yuwei Zhang, Qingyuan Lu, Kai Liu, Wensheng Dou, Jiaxin Zhu, Li Qian, Chunxi Zhang, Zheng Lin, and Jun Wei. CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge. *CoRR*, 2025. doi:10.48550/arXiv.2501.16155.
- 52 Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. LLM-Based Unit Test Generation via Property Retrieval. *CoRR*, 2024. doi:10.48550/arXiv.2410.13542.

A Mutation Operators

■ **Table 5** Mutation operators applied in the evaluation.

Category	Operator	Transformation
Arithmetic	cxx_minus_to_noop	$-x \rightarrow x$
	cxx_add_to_sub	$+ \rightarrow -$
	cxx_sub_to_add	$- \rightarrow +$
	cxx_mul_to_div	$* \rightarrow /$
	cxx_div_to_mul	$/ \rightarrow *$
	cxx_rem_to_div	$\% \rightarrow /$
Boundary	cxx_le_to_lt	$\leq \rightarrow <$
	cxx_lt_to_le	$< \rightarrow \leq$
	cxx_ge_to_gt	$\geq \rightarrow >$
	cxx_gt_to_ge	$> \rightarrow \geq$
Comparison	cxx_eq_to_ne	$== \rightarrow !=$
	cxx_ne_to_eq	$!= \rightarrow ==$
	cxx_le_to_gt	$\leq \rightarrow >$
	cxx_lt_to_ge	$< \rightarrow \geq$
	cxx_ge_to_lt	$\geq \rightarrow <$
	cxx_gt_to_le	$> \rightarrow \leq$
Logical	cxx_remove_negation	$!a \rightarrow a$

B Hardware and Tooling

■ **Table 6** Development servers and GPU cluster used for experiments.

Identifier	CPU/GPU Model	Cores	RAM
dev-1	Intel Xeon Gold 6448H ($\times 4$)	128	2 TB
dev-2	Intel Xeon Gold 6448H ($\times 2$)	64	1 TB
dev-3	Intel Xeon Platinum 8260	24	188 GB
gpu-1	NVIDIA H200	16 896	141 GB