CIplus
Band 1/2026

Evaluating the Effectiveness of LLM-Generated Unit Tests in the Context of Industrial DBMS

Vekilmuhammet Bekmyradov

II

Abstract

Software testing consumes considerable resources in industrial database management systems, where undetected faults can cause data loss and costly operational disruptions. While Large Language Models (LLMs) have demonstrated remarkable code-generation capabilities, existing evaluations mostly rely on benchmarks that are likely present in LLM training data. This makes it unclear whether reported performance reflects genuine reasoning or memorization. We provide empirical evaluation of LLMs on unit test generation in a large proprietary database system that is absent from public training corpora. We compare three commercial general-purpose and one open-weight coding model across two systems: the proprietary SAP HANA and the open-source LevelDB. The study employs two generation methodologies. The first is test amplification, where existing human-written test suites are extended. The second is whole-suite generation, where tests are synthesized from source code files. To address the compilation challenges inherent in complex C++ projects, we implement an iterative repair loop guided by compiler feedback, enabling models to resolve compilation errors over multiple attempts. Additionally, to assess the impact of project-specific context on generation quality, we evaluate model performance under two configurations: source code only and source code augmented with its depended header files. We adopt an evaluation framework comprising compilation success rate, code coverage, and mutation score to evaluate syntactic correctness and the fault-detection effectiveness of the generated tests. Our results reveal a substantial performance gap between open-source and proprietary software systems. On the LevelDB project, multiple models achieved near-perfect mutation scores up to 100%, surpassing the human-written baseline by up to 50 percentage points, while GPT-5 increased line coverage from 73.78% to 82.69%. In contrast, performance on SAP HANA degraded considerably. Our test amplification approach achieved 77.33% line coverage and a mutation score of 39.54%, while whole-suite generation with auxiliary context reached a maximum of 62.11% line coverage and 25.14% mutation score. Both approaches performed substantially below the human-written baseline (baseline values are confidential). The iterative repair loop increases compilation success rates by approximately 2–3× within the first few iterations, with GPT-5 reaching up to 99%. However, qualitative analysis reveals that as the number of repair iterations increases, models shift their optimization toward compilability at the expense of effectiveness. Including header files as auxiliary context in the prompt increased line coverage by 10–15 percentage points and mutation scores by 1.5–4.4x across all four models compared to source-only input.

Content

List of Tables

List of Figures

1. Introduction

Since the establishment of the theoretical foundations of computation by Alan Turing (Turing 1937), software engineering has undergone a series of transformative paradigm shifts aimed at increasing abstraction and automation. The first major leap from manual binary manipulation involved the adoption of assembly language, which replaced raw machine code with mnemonic abstractions (Washizaki 2024). This trajectory continued with the development of high-level languages, allowing developers to define logic closer to human reasoning rather than hardware constraints. Historically, each elevation in abstraction has lowered the barrier to entry to programming and accelerated development cycles (Washizaki 2024). Currently, the field is experiencing a new revolution driven by the emergence of Large Language Models (LLMs) based on the Transformer architecture (Vaswani et al. 2017). These models have reduced the cost and time required for software construction, capable of generating syntactically correct and complex code snippets in seconds that previously required significant human effort (M. Chen et al. 2021; Y. Li et al. 2022; R. Li et al. 2023). However, this rapid generation capability introduces some challenges: LLM-generated code can exhibit logical errors, memorisation of training data and security vulnerabilities (Al-Kaswan, Izadi, and Deursen 2024; Al-Kaswan 2024). As a result, the role of software testing has become more critical than ever to verify that these automatically generated solutions align with functional requirements (Junjie Wang et al. 2024; Schäfer et al. 2024).

In the domain of large, business critical systems such as industrial database management systems (DBMS), undetected faults can cause data loss, downtime, or cascading failures with substantial cost to organizations and users (Bach et al. 2022). Therefore, testing consumes a significant portion of development resources (Myers 2004), necessitating automated solutions to scale effectively.

In this work, we investigate the test generation performance of LLMs in the context of large-scale industrial DBMSs. Based on both our observations and supporting literature, we identify the following key challenges:

1. **Data Contamination and Validity:** The majority of existing research evaluates LLMs using open-source projects (e.g., HumanEval, Defects4J) that likely exist within the public training data of these models (M. Chen et al. 2021). Recent studies indicate that models often memorize solutions rather than generalizing to new problems (Y. Dong et al. 2024; W. Chen et al. 2025). This

leads to inflated performance metrics on familiar code. As a result, it remains unclear whether high performance on standard benchmarks translates to genuine reasoning capabilities on unseen industrial tasks (S. Chen, Pusarla, and Ray 2025).

2. **Complexity of Large C++ Projects:** Unlike Python or Java, C++ requires complex build configurations, manual memory management, and strict adherence to compilation units (Y. Zhang et al. 2025). Research has shown that LLMs struggle with these features, frequently generating code that fails to compile due to missing project-specific dependencies or incorrect usage of pointers and templates (Jian Wang et al. 2025). Generating correct C++ code that links successfully against a massive, proprietary codebase requires context awareness (e.g., header dependencies, compilation metadata) (Y. Zhang et al. 2025).
3. **Absence of Holistic Evaluation:** Many evaluations predominantly rely on code coverage as the sole proxy for test quality, ignoring the semantics of the generated tests (Junjie Wang et al. 2024). While recent literature has introduced specific metrics such as *pass@k* to measure the probability of generating at least one functionally correct solution (M. Chen et al. 2021), or *build@k* to quantify compilation success rates (Mundhra, Valk, and Izadi 2025), they are often applied in isolation. Similarly, while advanced techniques like mutation-guided test generation exist (Harman et al. 2025), they are rarely integrated into a unified evaluation framework. Thus, existing approaches often lack a holistic, multidimensional assessment of generation quality that simultaneously considers compilation, execution, and fault detection (Tong and T. Zhang 2024).

In this work, we evaluate LLM-generated unit tests in the context of an industrial, proprietary Database Management Systems (DBMS) SAP HANA and a smaller open-source project (LevelDB). Because SAP HANA is closed-source, it is guaranteed to be absent from the training data of public LLMs, providing a unique environment to assess the true generalization capabilities of the models (S. Chen, Pusarla, and Ray 2025). We compare four distinct models: GPT-5 (OpenAI 2025), Claude 4 (Sonnet) (Anthropic 2025), Gemini 2.5 Pro (G. Team 2025), and Qwen-3-Coder (Q. Team 2025). To mitigate the specific complexities of C++ projects, we design and implement an iterative generation pipeline with *Compiler-Feedback Repair Loop* to autonomously fix build errors.

We conducted the evaluation across two operational scenarios:

- **Scenario 1 (Test Amplification):** Extending an existing, partial test suite to improve coverage and mutation scores.

- **Scenario 2 (Whole-Suite Generation):** Generating a complete test suite from scratch using source code and its dependencies.

The primary contributions of this work are as follows:

1. **Zero-Contamination Evaluation:** We provide empirical evidence of LLM performance on a closed-source industrial system, isolating reasoning capabilities from memorization effects.
2. **Robust Generation Pipeline:** We propose a methodology for C++ test generation that integrates few-shot sampling and iterative self-repair, addressing the compilation challenges inherent in complex C++ projects.
3. **Comparative Analysis:** We present an empirical evaluation of three general-purpose proprietary models and one open-source coding model on two target systems (SAP HANA and LevelDB) across two scenarios (Test Amplification and Whole-Suite Generation).
4. **Multidimensional Metrics:** We propose and utilize an evaluation framework that extends beyond coverage to include Compilation Success Rate (CSR) and Mutation Score (MS).

This report is organized as follows: In Chapter 2, we provide the necessary background and related work on LLMs, software testing, and the systems under study. We detail the experimental framework, including the generation pipeline and test reduction heuristics, in Chapter 3 . In Chapter 4, we describe the evaluation methodology, the mutation analysis workflow, and the specific constraints of large-scale systems. We then present the quantitative results and analysis of our research questions in Chapter 5. In Chapter 6, we discuss the implications of our findings and address the limitations of the study. Finally, in Chapter 7, we summarize the findings and outline directions for future work.

2. Background

To provide the necessary context for this work, we present relevant literature and core concepts regarding LLMs and automated software testing in this chapter. In Section 2.2, we define key terms used in software testing and include subsections on code coverage metrics and mutation analysis as a measure of fault detection. Subsequently, in Section 2.3 we provide a survey of recent LLM-based test generation works and highlight gaps in current research. We then describe the architecture and specific constraints of the systems under study, the proprietary SAP HANA database and the open-source LevelDB project in Section 2.4. Finally, in Section 2.5 we examine the critical issue of data contamination in LLM evaluation for code generation tasks and explain why SAP HANA serves as a unique environment for this investigation.

2.1. Large Language Models

In this section, we provide the necessary background information on Large Language Models. We begin in Section 2.1 by introducing the architecture and operational principles of LLMs, exploring key prompt engineering techniques, and discussing inherent limitations such as hallucination.

2.1.1. Definition and Core Principles

Large Language Models (LLMs) are deep neural networks trained to predict the next token in a sequence given its preceding tokens. Most modern LLMs are based on the Transformer architecture, which relies on a self-attention mechanism to weigh the importance of different parts of the given input sequence and compute contextualized representations in parallel. This enables efficient training on large-scale datasets and effective use of contextual information (Vaswani et al. 2017).

"Large" in Large Language Models primarily refers to the order of magnitude of their learnable parameters, typically ranging from billions to trillions (W. X. Zhao et al. 2023). This massive parameter space allows these models to be trained on large corpora of text and source code, enabling them to capture intricate statistical regularities. These patterns extend beyond simple word co-occurrences to include complex syntactic

structures, long-range semantic dependencies, and idiomatic programming constructs (Kaplan et al. 2020).

Empirical studies have shown that increasing model size, training data, and computational resources lead to predictable improvements in language modeling performance, a phenomenon commonly described by *scaling laws* (Kaplan et al. 2020).

2.1.2. Prompt Engineering

Prompt engineering is defined as the systematic process of designing natural language inputs to condition a pre-trained language model for specific downstream tasks, without the need for updating the model's parameters (P. Liu et al. 2023). A well-constructed prompt typically comprises task instructions, domain-specific context, and optional examples. Prompting operates by aligning the downstream task with the model's pre-training objective (e.g., next-token prediction), effectively steering the generated probability distribution toward the desired output (Brown et al. 2020; Schulhoff et al. 2024).

Language models are sensitive to how a task is described; small changes in wording, example order, or formatting can produce significantly different outputs. The simplest technique is *zero-shot prompting*, which provides only an instruction to the model. However, Brown et al. (2020) demonstrate that adding a few examples often substantially improves performance. This is known as *in-context learning*, where task descriptions and examples provided directly within the input prompt strongly influence the generated output without any modification of the model's parameters (Q. Dong et al. 2024; Brown et al. 2020). Such behaviors appear only when models reach sufficient scale and are referred to as *emergent abilities* (Wei, Tay, et al. 2022). Another example is *chain-of-thought (CoT) reasoning*, the ability to decompose complex problems into intermediate steps (Wei, X. Wang, et al. 2022). While originally proposed as a few-shot technique, Kojima et al. (2022) subsequently demonstrated that LLMs are also *zero-shot reasoners*, capable of generating reasoning chains simply by appending the trigger phrase "Let's think step by step" (Zero-Shot CoT). In practice, these capabilities enable the adoption of general-purpose LLMs to new domains without expensive retraining. Beyond these, other commonly used techniques include instruction prompts, template-based prompts, and role-based prompts (Schulhoff et al. 2024; P. Liu et al. 2023).

Prompt structure can also enable adversarial behavior ("jailbreaking"), where carefully crafted prompts evade model constraints. Empirical studies document both the vulnerability and possible defenses (Y. Liu et al. 2023).

In this work, we employ several prompting strategies, including zero-shot, few-shot, instruction-based, and role-based techniques. To ensure a fair evaluation of model performance, we standardize the prompt templates across all experimental runs, keeping the instructional wording and structural format invariant while only varying the specific input code. All prompt templates and representative examples are documented in Appendix A.

2.1.3. Limitations and Risks

Despite their capabilities, LLMs have important limitations. Their outputs are inherently stochastic. Repeated generations from the same prompt can produce different results (Bender et al. 2021). Furthermore, LLMs may generate statements that appear plausible but are factually incorrect or unsupported. This is commonly referred to as *hallucination* (L. Huang et al. 2025). These issues arise from the probabilistic nature of the training objective and from biases and gaps in the training data.

2.2. Software Testing

Software testing is a software engineering activity used to increase confidence that software behaves as intended and to detect faults before deployment (Myers 2004; Ammann and J. Offutt 2008). Testing provides empirical evidence about program behavior under given conditions, but it does not formally verify correctness. It reduces the risk of deviations from expected behavior. Economically, the investment in software testing is justified by the significant financial impact of software faults, which can lead to system failures and costly operational disruptions if left undetected (Myers 2004; Ammann and J. Offutt 2008; Bach 2022).

Testing is commonly organized into hierarchical levels that differ in scope, degree of isolation, and execution cost (Myers 2004; Ammann and J. Offutt 2008; Bach 2022). Unit tests typically exercise individual units of code, such as a function or a class, in isolation (Washizaki 2024). Component or module tests consist of collections of units assembled into a file, package, or class, and therefore cover a larger scope. Integration tests check interactions between components. System tests evaluate the complete system in an environment that resembles production (Bach 2022).

In this work, we use the terms *unit test*, *test suite*, *component test*, and *subcomponent test*. Prior work often employs the term *test case* to denote an individual test (Washizaki 2024; Bach 2022). In this work, we use the term *unit test* to refer to this atomic testing entity. Within the scope of this work, a *test suite* denotes a collection of one or more

unit tests, a *component* or *subcomponent* denotes a logical grouping of one or more test suites. This logical grouping of the tests is illustrated in the Figure 2.1.

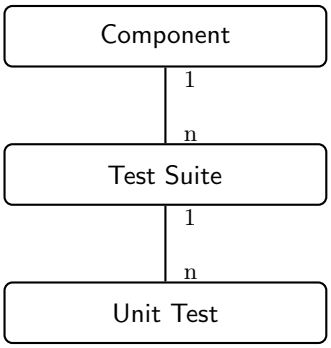

Figure 2.1.: Logical grouping of tests in this work. Inspired by (Bach et al. 2022)

2.2.1. Code Coverage

Code coverage measures which parts of a program are executed when a test suite runs (Bach 2022). The basic idea is to run the tests on the program, record the source regions that are executed, and calculate the fraction of covered elements over the total. Coverage reports the execution of the code, not the correctness of its behavior.

Common coverage criteria include line coverage, branch coverage, and function coverage. Line coverage counts executed statements. Branch coverage checks whether each branch of a conditional statement has been taken. Function coverage measures whether every function in the codebase has been called at least once by the test suite (Bach 2022; Hutchins et al. 1994; Gupta and Jalote 2006). There are other types of coverage metrics, such as *modified condition/decision coverage* (MC/DC) or path coverage, which are more complex (Chilenski and Miller 1994; Bach 2022), but they are not used in this work. Listing 2.1 and Listing 2.1 provide examples for line and branch coverage in C++.

Coverage is easy to compute and understand. It helps find untested code regions and is useful for tracking test progress over time. Code coverage is commonly monitored within automated continuous integration (CI) pipelines to provide immediate feedback on the scope of the test suite (Santos et al. 2024). Coverage also has clear limits. High coverage does not guarantee that tests detect faults. Tests can execute code without checking correct behavior. In particular, coverage can be inflated by trivial or assertion-less tests (J. Chen et al. 2017; Schäfer et al. 2024).

```
1  int compute(int x) {
2      int a = 10;              // Executed
3      int b = 20;              // Executed
4      if (x < 0)
5          return a;            // Executed if x < 0
6      int c = a + b;           // NOT executed if x < 0
7      return c;                // NOT executed if x < 0
8  }
9
10 int main() {
11     compute(-1);             // Only early return path executed
12 }
```

Listing 2.1: Example illustrating line coverage in C++

```
1  int sign(int x) {
2      if (x > 0) {                 // True branch
3          return 1;
4      } else {                     // False branch
5          return 0;
6      }
7  }
8
9  int main() {
10     sign(5);                     // Only true branch exercised
11 }
```

Listing 2.2: Example illustrating branch coverage in C++

2.2.2. Mutation Testing

This subsection defines the core concepts of mutation testing, discusses its advantages over traditional code coverage, and outlines the challenges inherent in its practice. Finally, we introduce mutation testing tool employed for mutation analysis in this study.

Definition and Terminology

To understand the mechanics of mutation testing, it is necessary to distinguish between standard testing concepts. An *error* is a human mistake (e.g., a typo) that results in a *fault* in the source code. If executed, this fault may manifest as a system *failure* (Ammann and J. Offutt 2008). Mutation testing is a fault-based technique

that evaluates test quality by systematically seeding artificial changes referred to as *mutants* into the program code. It checks whether the test suite can detect those mutants (DeMillo, Lipton, and Sayward 1978; Jia and Harman 2011). Each mutant represents a small syntactic change, such as replacing '+' with '-' or modifying a relational operator. Such changes are generated using predefined *mutation operators*, which specify how the original code is altered.

Figure 2.2 illustrates the example of simple mutation analysis. A mutation operator changes the boundary condition ($>\rightarrow\geq$). A generic test case like `age=20` cannot distinguish the behavior, therefore the mutant survives. Only a boundary test case like `age=18` exposes the fault, killing the mutant.

Original Code

```
bool isValid(int age) {
// Check if adult
    if (age > 18) {
        return true;
    }
    return false;
}
```

Mutated Code

```
bool isValid(int age) {
// Operator Mutated: > to >=
    if (age >= 18) {
        return true;
    }
    return false;
}
```

Test Case Analysis:

- `test(20)`: Returns `true` for both. **Mutant Survives.**
- `test(18)`: Returns `false` (Original) vs `true` (Mutant). **Mutant Killed.**

Figure 2.2.: Illustration of mutation analysis.

Mutation operators define the rules by which mutants are generated. Common operator classes include arithmetic operator replacement (AOR), relational operator replacement (ROR), logical connector replacement (LCR), absolute-value insertion (ABS), unary operator insertion (UOI), statement deletion (SDL), and constant replacement (CRP) (A. J. Offutt et al. 1996; Papadakis et al. 2019). The chosen operator set affects both the cost of mutation testing and the types of faults that can be simulated. Prior work has shown that a small, well-chosen operator subset can retain much of mutation testing's effectiveness while reducing cost. Operator selection is an active area of practical research (A. J. Offutt et al. 1996; Delamaro et al. 2014; Kaufman et al. 2022; Petrovic et al. 2022). We document the mutation operators used for this work in Appendix C

Under the assumption that the test suite passes on the original program, a mutant is

said to be *killed* if its execution causes at least one test case to fail (Jia and Harman 2011). Conversely, if the test suite passes on the mutant, it is referred to as a *survived mutant*. The *mutation score* is defined as the ratio of killed mutants to the total number of non-equivalent mutants. A *non-equivalent mutant* is a mutant whose behavior is semantically different from that of the original program code (Jia and Harman 2011; Madeyski et al. 2014).

Advantages over Code Coverage

Code coverage reports which code was executed, but it does not indicate whether the executed behavior was checked by meaningful *assertions*. Assertions are statements within the test that explicitly validate whether the actual output matches the expected result. A test suite may therefore achieve high coverage while still lacking the checks necessary to detect *fault patterns*, such as logic errors or boundary condition failures, due to weak or missing assertions. Mutation testing complements coverage by measuring whether the existing tests actually detect injected faults, exposing weaknesses that coverage alone may miss (Jia and Harman 2011; Frankl, Weiss, and Hu 1997; Papadakis et al. 2019).

Mutation testing can reveal missing assertions and weak *oracles* that coverage can not (J. Chen et al. 2017; Papadakis et al. 2019). A *test oracle* is a mechanism that determines whether a program execution produced a correct output based on the input of a test. Oracle specification is a well-known challenge in software testing research, also referred to as *test oracle problem* (Barr et al. 2015).

Practical Challenges

Despite its benefits, mutation testing has practical limitations that prevent wide adoption. First, it is computationally expensive because many mutants must be generated, compiled, and executed, which increases testing time and resources (Jia and Harman 2011; Papadakis et al. 2019; Petrovic et al. 2022). Second, the equivalent-mutant problem requires manual or heuristic effort. Identifying such mutants is undecidable in general (Delgado-Pérez and Chicano 2022; Madeyski et al. 2014; Tian et al. 2024). Third, mutation tools must integrate with real build systems and handle language-specific and optimization-related issues. Engineering robust integration for large, compiled systems is nontrivial and adds to adoption cost (Denisov and Pankevich 2018; Papadakis et al. 2019). Empirical studies report mixed cost–benefit tradeoffs. Mutation testing is a strong indicator of test quality, but is generally more costly than simple coverage-based metrics, which limits its common use in many industrial contexts (Frankl, Weiss, and Hu 1997; Jia and Harman 2011).

The Mull Framework

For this work we use Mull (Denisov and Pankevich 2018)[1], an open-source mutation testing tool based on the LLVM framework[2]. Mull operates on LLVM intermediate representation (IR), which enables language independence for any language that compiles to LLVM IR, including C and C++ (Denisov and Pankevich 2018). Mull modifies and recompiles only the affected IR fragments. Therefore, it reduces the overhead of recompiling entire programs.

2.3. LLMs for Test Generation

Large language models trained on source code have demonstrated substantial capability to generate and reason about code (M. Chen et al. 2021). Early specialized models, such as Codex, marked a significant milestone in program synthesis by introducing methods to generate multiple candidate solutions and select the most viable ones. This increased the likelihood of producing correct code (M. Chen et al. 2021; Y. Li et al. 2022). Subsequent models with publicly available parameters, such as Code Llama and StarCoder, provided accessible, high-quality baselines that have accelerated research and replication (Rozière et al. 2023; R. Li et al. 2023).

Empirical studies consistently report that LLMs produce natural, human-like tests, but substantial fraction of generated tests fail to compile or execute correctly without post- processing, additional context, or iterative repair (Z. Yuan, M. Liu, et al. 2024; Siddiq et al. 2024; Yang et al. 2024; Du et al. 2024; Dakhel et al. 2024). Works that measure mutation scores, execution pass rates and developer acceptance alongside coverage reports showed different conclusions than coverage-only studies, indicating that having only coverage is an insufficient metric for test usefulness in LLM-generated tests (Dakhel et al. 2024; Harman et al. 2025; Junjie Wang et al. 2024).

Some literature proposes methods to improve the effectiveness of generated tests. For example, iterative self-revision of model outputs improves compilability and assertion correctness (Z. Yuan, M. Liu, et al. 2024). Enhancing prompts with context derived from static analysis, such as class hierarchies, function signatures, and header dependencies significantly improve the likelihood of successful compilation and reduce hallucinated references (Pan et al. 2025; Y. Zhang et al. 2025). Furthermore, integrating feedback loops that utilize compilation errors or mutation analysis results (i.e., prompting the model to specifically target surviving mutants) enhances the functional quality and fault-detection capability of the generated tests (Dakhel et al. 2024; Harman et al.

[1] https://github.com/mull-project/mull
[2] https://github.com/llvm/llvm-project

2025). These strategies are complementary in practice and yield greater improvements when integrated.

Prior work suggests that LLMs show strong performance on Python programming language but struggle to generalize effectively across other programming languages (Zan et al. 2025). Another study reports a significant performance gap in LLM-based automated program repair when addressing C/C++ faults, compared to their success on the Defects4J benchmark for Java (Jian Wang et al. 2025). These findings may explain why substantially less empirical research exists for LLM-based test generation in low-level systems compared to high-level languages. Low-level, compiled languages such as C and C++ involve complex build systems and native dependencies, which make compilation and linking non-trivial. More information such as project dependency graphs and build metadata needs to be incorporated into generation pipelines. In addition, undefined behavior and manual memory management increase the risk of spurious crashes or flaky tests (Berndt, Bach, and Baltes 2024; Berndt, Baltes, and Bach 2024). These characteristics imply that language-aware heuristics and deeper program analysis are often required to obtain usable tests for such systems (Y. Zhang et al. 2025; Jian Wang et al. 2025; Zan et al. 2025; Mundhra, Valk, and Izadi 2025).

Industry case studies confirm that LLMs can add practical value when paired with verification, strict filtering, governance, and human review (Alshahwan et al. 2024; Dakhel et al. 2024). But they also show that post-generation validation is essential to catch hallucinated or unsafe edits. Also, in large industrial systems, building and integration constraints as well as domain-specific practices strongly affect deployment feasibility (Y. Dong et al. 2024; Zan et al. 2025; Harman et al. 2025; Mundhra, Valk, and Izadi 2025).

2.4. Systems Under Study

2.4.1. SAP HANA

SAP HANA is a commercial, in-memory database management system (DBMS) developed and maintained by SAP. It is a large, performance-oriented database management system, implemented primarily in C and C++, and deployed in a variety of industrial settings where correctness and performance are essential (Bach 2022).

SAP HANA is a multi-component system whose components are interconnected, forming a complex dependency structure that includes internal application programming interface (API) calls, shared libraries, and link-time dependencies. The codebase consists of approximately 40 million lines of code, with several decades of development history (Bach et al. 2022). The testing process of SAP HANA involves multiple stages.

For example, pre-submit checks, post-submit testing, and longer-running continuous testing that together aim to balance fast developer feedback with thorough regression detection (Bach et al. 2022; Bach 2022). The testing stages of SAP HANA are illustrated in Figure 2.3. The test code of SAP HANA is organized in test suites, each containing between 1 to 20 000 test cases. The total number of tests executed from pre-submit testing to the final release is estimated to be over 900000. Executing all of them sequentially would require up to 4 weeks (Bach et al. 2022). More information on the project can be found in Appendix B.

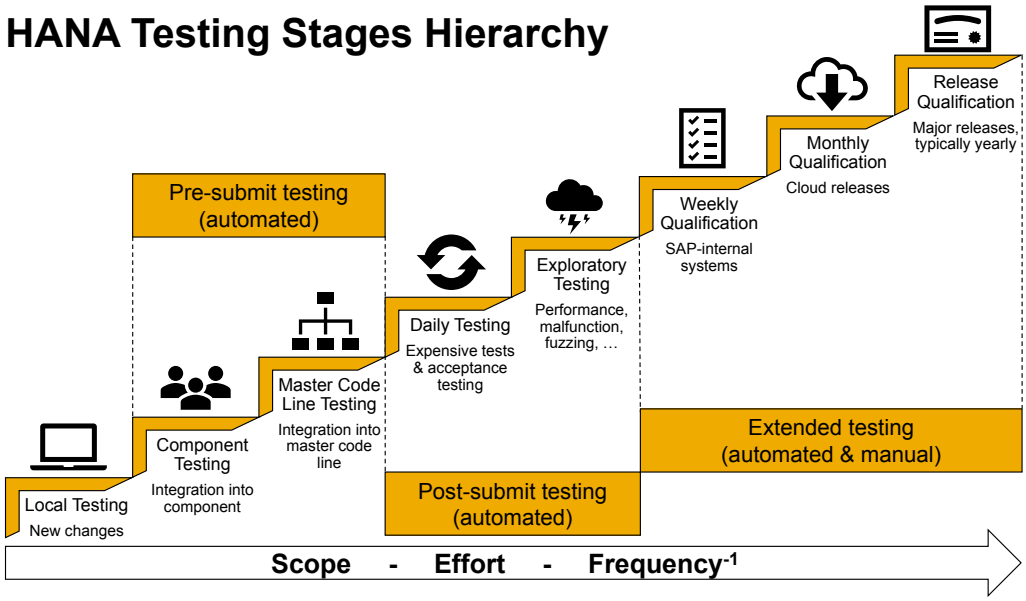

Figure 2.3.: The testing stages of SAP HANA (Bach 2022).

Confidentiality

Due to confidentiality agreements protecting the proprietary SAP HANA source code, absolute baseline values such as code coverage and mutation score cannot be disclosed. Results relative to this baseline are therefore reported qualitatively. All other experimental results are presented in absolute terms unless otherwise stated.

2.4.2. LevelDB

To compare the performance of LLMs on an accessible, open-source system before applying them to the proprietary environment, this study employs LevelDB[3], a key-value storage library developed by Google. LevelDB is implemented in C++ and provides an ordered mapping from string keys to string values.

LevelDB was selected as the baseline system for three primary reasons:

1. **Technological Alignment:** It shares the same core technology stack as SAP HANA. It is written in C++ and uses GoogleTest framework[4]. This allows for a direct comparison.
2. **Open Source:** As an open-source project, it allows for reproducible experiments and transparent analysis of the LLM's training data exposure.
3. **Manageable Complexity:** Its moderate size enables rapid iteration of the generation pipeline and facilitates manual verification of generated tests during the development phase.

We analyze a version of the LevelDB repository comprising 133 total code files, including 76 C++ source files (`.cc`), 27 of which are test files. Based on `cloc`[5] analysis, the codebase contains approximately 21,207 lines of code (LOC). The project has 27 test suites with a total of 211 individual test cases. Detailed information regarding the specific project versions used in this study can be found in Appendix B.

2.5. Evaluation on Closed-Source Systems

A challenge in evaluating Large Language Models on code generation tasks is the risk of *data contamination* (Riddell, Ni, and Cohan 2024). Modern LLMs are pre-trained on internet-scale repositories of open-source code (e.g., GitHub). Therefore, evaluations conducted on public projects often fail to distinguish between the model's ability to reason about new problems and its ability to recall memorized solutions (Y. Dong et al. 2024).

SAP HANA presents a unique evaluation environment in this context. As a proprietary, closed-source system, its codebase is not present in the public pre-training corpora of models. This exclusion guarantees that the models have effectively zero prior exposure

[3] https://github.com/google/leveldb
[4] https://github.com/google/googletest
[5] https://github.com/AlDanial/cloc

to the specific internal APIs, macro definitions, and architectural patterns of the system.

This study evaluates the *zero-shot* and *few-shot* generalization capabilities of LLMs on unseen data. This imposes two distinct implications for our methodologies: memorization. Successful test generation demonstrates the model's generalization on an unseen tasks.

1. **Validity of Evaluation:** High performance on this task cannot be attributed to memorization. Successful test generation demonstrates the model's generalization on unseen tasks and its capability to perform in-context learning.
2. **Contextual Necessity:** The absence of implicit knowledge requires explicit context management. Therefore, all necessary domain knowledge, such as dependency structures, test fixtures, and namespaces, must be injected dynamically via prompt engineering or retrieval mechanisms.

3. Study Design and Test Generation

In this chapter, we outline the research questions guiding this study, describe the Large Language Models selected for the experiments, and provide a detailed explanation of the applied methodology.

3.1. Research Questions

To systematically evaluate the capabilities of LLMs in the context of C++ test generation, we structure our investigation around four research questions.

Previous study pointed out that code coverage is not strongly correlated with a test suite's ability to detect actual faults (Inozemtseva and Holmes 2014). To address this limitation, we complement code coverage metrics with mutation analysis, which evaluates the test suite's ability to detect and reject artificial faults.

> **RQ1:** *To what extent can LLMs generate effective tests in terms of code coverage and mutation score for proprietary, large-scale systems like SAP HANA?*

A critical concern in LLM evaluation is *data contamination*, where benchmarks derived from open-source repositories (e.g., GitHub) appear in the model's training data, leading to artificially inflated performance metrics (Riddell, Ni, and Cohan 2024). To assess generalization of the models' test generation capability on SAP HANA, a system whose source code the models have never seen during training. Motivated by this distinction, we pose the following research question:

> **RQ2:** *Do LLM-generated tests differ in terms of code coverage and mutation score when evaluated on open-source versus closed-source systems.*

LLM-generated code may include calls to non-existent APIs, incorrect type usage, or syntax errors (Z. Yuan, M. Liu, et al. 2024). While recent works suggest that *self-repair* mechanisms can resolve some of these issues (Olausson et al. 2023), the efficacy of such iterative feedback loops in a strictly typed C++ environment, where compiler errors can be verbose and complex, remains under-explored (Cassano et al. 2023). To

quantify the effectiveness of such a self-repair mechanism, we investigate the following question:

> **RQ3:** *To what extent does the iterative compiler-feedback loop improve the compilation success rate of generated tests compared to one-shot generation?*

C++ development relies heavily on cross-file dependencies and header definitions for type safety. Recent research indicates that withholding this repository-level context degrades model performance (Shrivastava et al. 2023). We hypothesize that providing header files within SAP HANA context is necessary for the model to construct valid objects and invoke complex APIs correctly. To empirically validate this hypothesis, we address the following research question:

> **RQ4:** *Does the inclusion of auxiliary context, such as header files, improve the code coverage and mutation score of the generated tests for SAP HANA?*

3.2. Studied LLMs

We evaluated four models in this study: one open-weight code-specialized model and three proprietary general- purpose models. The open-weight model is the 128,000 context window version of Qwen3-Coder (`Qwen3-Coder-480B-A35B-Instruct`)[1] (Q. Team 2025). The proprietary models are OpenAI's GPT-5 (OpenAI 2025), Anthropic's Claude 4 (Sonnet) (Anthropic 2025), and Google's Gemini 2.5 Pro (G. Team 2025). Access to the proprietary models was provided via internal SAP APIs.

All models were configured with temperature = 0 to reduce non-deterministic outputs from LLMs (W. X. Zhao et al. 2023; Brown et al. 2020). Table 3.1 shows the model specifications.

Model	Release Date	Parameters	Context Window
Qwen3-Coder	July 2025	480B (35B active)	128,000
GPT-5	Aug 2025	–	400,000[2]
Claude 4 (Sonnet)	May 2025	–	200,000[3]
Gemini 2.5 Pro	June 2025	–	1,000,000[4]

Table 3.1.: Large language models used in this study. '–' denotes values that were not publicly disclosed.

[1] https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct
[2] https://platform.openai.com/docs/models/gpt-5
[3] https://platform.claude.com/docs/en/about-claude/models/overview
[4] https://ai.google.dev/gemini-api/docs/models/gemini

3.3. Evaluation Metrics

To provide a multidimensional assessment of the LLM-generated tests, we evaluate performance across three primary categories: syntactic and semantic correctness, test quality, and operational efficiency.

3.3.1. Syntactic Correctness

Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the set of all test code files resulting from generation for a given experiment. Each file $f \in \mathcal{F}$ contains one or more unit tests. We define the subset of successfully compiled files as $\mathcal{F}_{comp} \subseteq \mathcal{F}$, such that a build artifact exists for every $f \in \mathcal{F}_{comp}$.

From the set of successfully compiled files $\mathcal{F}_{comp}$, we extract the set of individual executable unit tests $\mathcal{T}$. We define the subset $\mathcal{T}_{pass} \subseteq \mathcal{T}$ as the set of tests that execute without assertion failures or runtime errors.

Based on these sets, we define the correctness metric as follows:

- **Compilation Success Rate (CSR):** The proportion of test code files that are syntactically valid and successfully compiled by the build system.

$$CSR = \frac{|\mathcal{F}_{comp}|}{|\mathcal{F}|} \tag{3.1}$$

3.3.2. Test Quality Metrics

We assess the quality of the valid tests ($\mathcal{T}_{pass}$) using code coverage criteria and mutation analysis.

- **Code Coverage:** We report coverage according to two metrics. The higher these metrics are, the better:
  - **Line Coverage (Cov_L):** The percentage of source code lines executed by the test suite.
  - **Branch Coverage (Cov_B):** The percentage of control flow branches executed.

- **Mutation Score (MS):** Let $\mathcal{M}$ be the set of all mutants generated for the target component. Let $\mathcal{M}_{killed} \subseteq \mathcal{M}$ be the subset of mutants detected (killed) by the generated tests $\mathcal{T}_{pass}$. The mutation score is defined as:

$$MS = \frac{|\mathcal{M}_{killed}|}{|\mathcal{M}|} \tag{3.2}$$

  A higher MS indicates a stronger capability of the generated tests to detect faults.

3.4. Methodology

This section details the experimental setup designed to evaluate the efficacy of LLMs in generating unit tests for an industrial-scale DBMS. We categorize the evaluation into two primary operational scenarios that reflect real-world software engineering challenges:

1. **Test Amplification (Scenario 1):** The process of extending an existing, partial test file to cover untested functionality or edge cases. This involves exploiting the knowledge embedded in human-written tests to generate improved or complementary test cases (Kessel and Atkinson 2019).
2. **Whole-Suite Generation (Scenario 2):** The process of generating an entire test suite from scratch based solely on implementation artifacts. In our experiments, we discard existing tests and treat the system as a clean slate to evaluate the model's ability to synthesize complete test suites.

3.4.1. Generation Pipeline

We employ a unified generation pipeline that integrates LLM inference with an automated build and repair loop. The process is illustrated in Figure 3.1.

It is important to note that this process is not a single-shot execution. The pipeline defined in Figure 3.1 is executed iteratively for every file (test files for Scenario 1, source files for Scenario 2) until the entire project is processed.

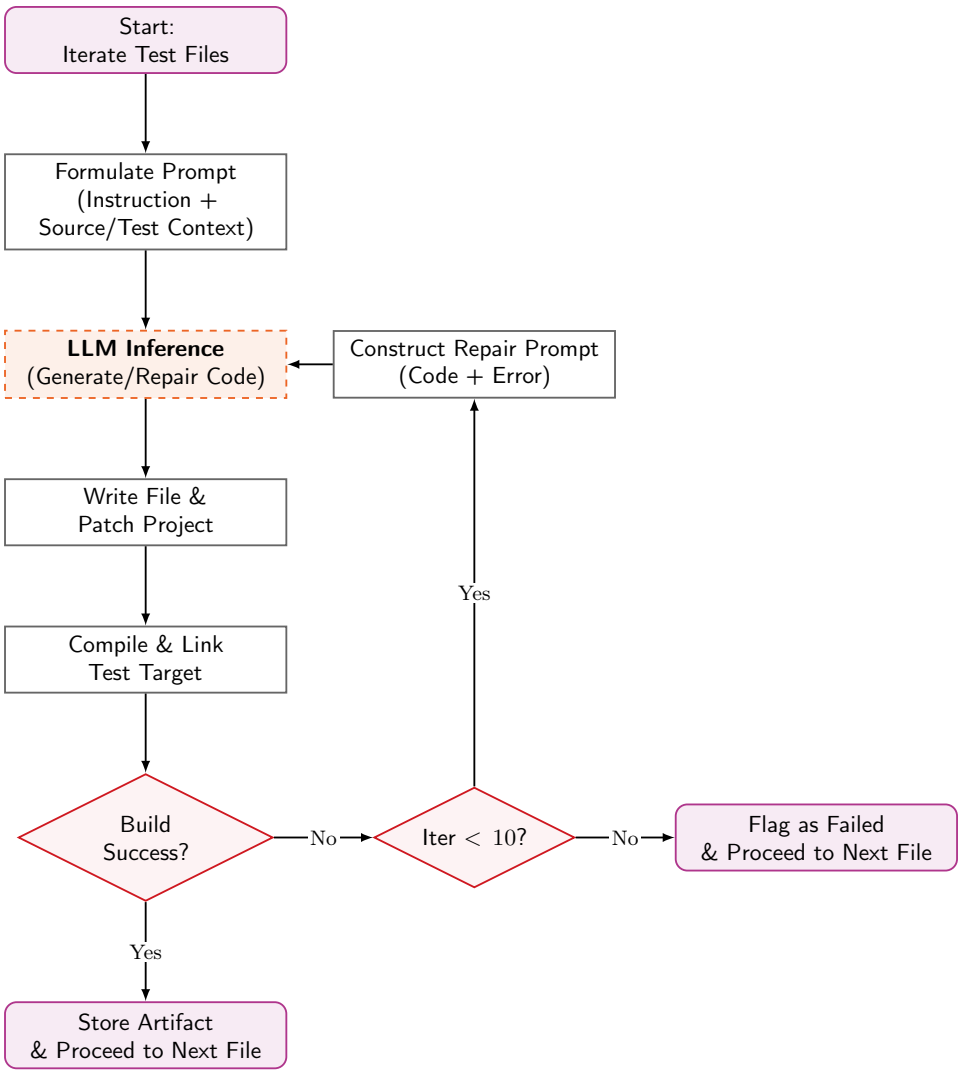

Figure 3.1.: Generation pipeline.

Detailed steps of the pipeline illustrated in Figure 3.1 proceed as follows:

1. **Context Preparation:** The relevant artifacts (existing test file content for Scenario 1; source code file and headers for Scenario 2) are retrieved and injected into the prompt.
2. **LLM Inference:** After the prompt is constructed, it's sent to the LLM. The model response is parsed to extract the C++ code blocks.

3. **Integration into the System:** The generated code is written to a new file and registered with the build system of the target project.
4. **Validation & Self-Repair:** The system attempts to compile the new target:
    - If compilation succeeds ($exit\_code = 0$), the artifacts are stored for evaluation.
    - If compilation fails, the compiler error log and the generated code is fed back to the LLM for a repair. This loop repeats until success or a maximum of $k = 10$ iterations is reached.

We selected the limit of $k = 10$ based on preliminary experiments with a larger limit ($k = 20$). We found that almost all compilable code was generated within the first 10 iterations. In fact, most useful results appeared within just 5 iterations, as later attempts often produced empty test bodies or trivial assertions. This can be explained by the model prioritizing compilation over code quality. We kept k at 10 to maintain consistency with our initial setup.

3.4.2. Scenario 1: Test Amplification

In this scenario, we investigate the model's ability to improve an existing, but partial, test suite. Since the target systems possess high baseline line coverage (90.21% for LevelDB), we artificially reduce the number of unit tests within the test suites. This is done to create "coverage gaps" for the LLM to fill. We chose this approach because increasing coverage at such high levels is harder than improving it in lower ranges. This is because remaining uncovered code often consists of complex error handling, defensive programming checks, or unreachable states, which are difficult to target without specific guidance (Ivankovic et al. 2019).

Automated Test Reduction Heuristic

We define a reduction process that operates at the Abstract Syntax Tree (AST) level using Tree-sitter[5]. This ensures that we remove complete unit test functions rather than arbitrary lines of code.

Let S be a test suite containing a set of unit tests $\{t_1, t_2, \ldots, t_n\}$. We define a reduction function $k_{remove}(S_i, \rho)$ that calculates the number of tests to be removed based on a target reduction percentage ρ.

[5] https://tree-sitter.github.io/

$$k_{remove}(S_i, \rho) = \begin{cases} 0 & \text{if } |S_i| \leq 1 \\ \min\left(|S_i| - 1,\ \max\left(1, \left\lfloor |S_i| \cdot \frac{\rho}{100} \right\rfloor\right)\right) & \text{if } |S_i| > 1 \end{cases} \tag{3.3}$$

where:

- S_i denotes the i-th test suite within a source file.
- $|S_i|$ is the number of atomic unit tests of that suite.
- ρ is the target reduction percentage (e.g., 90).
- $\lfloor \cdot \rfloor$ denotes the floor function.

This reduction heuristic is applied to every test suite across the entire set of test files within the target component.

The above heuristic ensures two critical criteria:

1. **Non-Emptiness:** For any suite with existing tests, at least one test is preserved to serve as a few-shot reference for the LLM. Thus, $|S_{reduced}| \geq 1$, where $|S_{reduced}|$ represents the test suite after reduction.
2. **Minimum Perturbation:** For any suite with $|S_i| > 1$, at least one test is effectively removed to enforce a generation task.

Selection of Reduction Baseline: SAP HANA

We configured two distinct reduction baselines: 90% and 99%. We specifically selected these aggressive reduction thresholds because lower reduction rates did not result in a substantial drop in coverage. For instance, in preliminary trials on SAP HANA, a 40% reduction still yielded a line coverage of 90.10%. This indicates that multiple tests execute the same code paths. As a result, such high residual coverage leaves negligible room for the LLM to demonstrate improvement. By reducing the suite by 90% and 99%, we ensure a sufficient coverage gap to effectively measure the model's generative capability.

Selection of Reduction Baseline: LevelDB

During the preliminary analysis of LevelDB, we observed that specific test suites, most notably `DBTest`, function as a comprehensive test suite exercising the code across the entire codebase. The `DBTest` suite alone achieved a line coverage of 79.32% and killed nearly 99% of the generated mutants (total of 1773), skewing the baseline metrics. Including such test suites would compromise the experimental validity of our unit test generation evaluation. As a result, we excluded such test suites to establish a cleaner, unit-focused human baseline. After this adjustment, the reference line coverage and mutations score for LevelDB are 73.78% and 52.8%, respectively.

For the test amplification methodology applied to LevelDB, we restrict our evaluation exclusively to the 99% reduction level. An initial comparison revealed that the starting line coverage difference between the 90% reduced suite (57.04%) and the 99% reduced suite (56.83%) was negligible (less than 1%). Given this small difference, we selected the 99% reduction case as the representative scenario for the LevelDB experiments.

Table 3.2 presents the coverage and mutation metrics for these reduced baselines compared to the original snapshot. Note that "Test Functions" refers to distinct test bodies obtained parsing the AST of a test file, distinguishing them from runtime instantiations. In this scenario, our evaluation consists of 130 test files for SAP HANA and 20 test files from LevelDB. "Original (all)" for LevelDB in the Table 3.2 refers to the snapshot before the exclusion of `DBTest` like test suites.

System	Configuration	Test Functions	Cov_L (%)	Cov_B (%)	MS (%)
SAP HANA Component	90% Reduction	382	71.90	39.75	34.48
	99% Reduction	253	66.71	35.33	30.41
	Original	2,323	–	–	–
LevelDB	99% Reduction	23	54.87	37.59	37.3
	Original	135	73.78	57.08	52.8
	Original (all)	227	90.21	78.18	100

Table 3.2.: Baseline Metrics for SAP HANA and LevelDB. '–' denotes values that were not publicly disclosed.

3.4.3. Scenario 2: Whole-Suite Generation

In the second approach, we discard existing tests entirely and task the LLM with generating a new test suite based solely on the implementation code. Focusing exclusively on non-test code files. From the SAP HANA component, we use 126 source files (`.cpp`). All 126 files combined have 276 unique header dependencies (`.hpp`, `.h`),

whereas from LevelDB repository, we use 37 source files (`.cc`) that use 44 unique header files (`.h`).

Context Seeding

As discussed in Section 2.5, LLMs lack prior exposure to the internal structure of SAP HANA, including its software architecture, dependency graphs, and namespace conventions. Consequently, we observed that providing raw source code alone frequently caused models to hallucinate incorrect build configurations or omit crucial project-specific headers, leading to poor generation results.

To mitigate this, we employ a *Context Seeding* strategy. This approach leverages the *few-shot learning* capabilities of LLMs (Brown et al. 2020) by providing a valid example within the prompt context. In addition to the source code, we inject the first 50 lines of an arbitrary valid test file from the project into the prompt. This provides the model with implicit knowledge of:

- Required include paths and library usages.
- The specific test fixtures and namespace conventions used in SAP HANA.
- Project-specific utility classes required for test setup.

We evaluate two configurations within this scenario:

1. **Source-only:** The prompt contains the content of the `.cpp` file.
2. **Source + Headers:** The prompt is injected with the current `.cpp` and the corresponding `.hpp` and `.h` files content to provide full API visibility.

Both configurations include the test code snippet in the prompt.

3.5. Context Window Limit

Despite LLMs possessing extensive context windows, spanning hundreds of thousands to millions of tokens (Reid et al. 2024), we frequently encountered scenarios where the context exceeded these architectural limits. This challenge was particularly obvious when providing the full set of header dependencies alongside the source C++ code. In the massive-scale environment of SAP HANA, a single source file can contain over several thousands of lines of code, often pulling in dozens of heavy header files that collectively surpass the model's input capacity. To mitigate these overflows without completely discarding such complex cases, we employed a truncation strategy that

systematically removes excess lines from the dependency headers, ensuring the prompt fits within the context window.

3.6. Experimental Setup

We used LangChain framework[6] as the abstraction layer to request the LLMs and manage prompt templates. For compilation and coverage measurement, we used the LLVM/Clang toolchain (version: 19.1.6).[7] Mutation testing was performed with the Mull framework (Denisov and Pankevich 2018).

The Qwen3-Coder model was served using the vLLM library[8] (Kwon et al. 2023) on an SAP-provided GPU cluster. The cluster consists of eight NVIDIA H200 GPU devices[9], with each device possessing approximately 143,771 MiB of memory.

Development, compilation, execution, and mutation testing were executed on three Linux servers running SUSE Linux Enterprise Server 15, all provided by SAP. Table 3.3 lists the hardware parameters for these machines.

Identifier	CPU model	Cores	RAM
dev-1	Intel(R) Xeon(R) Gold 6448H	256	2 TB
dev-2	Intel(R) Xeon(R) Gold 6448H	128	1 TB
dev-3	Intel(R) Xeon(R) Platinum 8260 @ 2.40GHz	48	188 GB

Table 3.3.: Development servers used for experiments.

3.6.1. Experiment Repetition

To account for the non-deterministic nature of LLMs (Ouyang et al. 2025; J. Yuan et al. 2025), we executed each experimental configuration twice. While we acknowledge that two repetitions ($N = 2$) are insufficient for statistical significance, this repetition serves as a sanity check to detect major deviations in model performance. This limitation was necessary due to the high computational cost of the experiments, where a single run requires days of computation to complete.

[6] https://github.com/langchain-ai/langchain
[7] https://github.com/llvm/llvm-project
[8] https://github.com/vllm-project/vllm
[9] https://www.nvidia.com/en-us/data-center/h200

4. Evaluation

In this chapter, we outline the specific challenges inherent in evaluating SAP HANA and detail the evaluation methodology employed in this study.

4.1. Evaluating SAP HANA

The evaluation of test effectiveness at an industrial scale, particularly within a large Database Management System like SAP HANA introduces significant computational and operational overhead. While the execution time of a single unit test is typically measured in fractions of a second (Bach 2022), the cumulative requirements for coverage instrumentation and mutation analysis scale non-linearly with the size of the codebase (Jia and Harman 2011).

4.1.1. Computational Overhead

According to Bach (2022), a full coverage analysis run for the SAP HANA requires approximately 1877 hours or 78.20 days of test execution, generating over 130 GB of raw coverage data. 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) (Bach 2022).

The sheer scale of the SAP HANA codebase makes mutation testing on the entire project impractical for the scope of this study. Performing mutation analysis on one of the test targets required two full days of computation on a 48-core machine with 188 GB of memory, and eventually failed after exceeding the machine's memory limit. Therefore, we adopted an incremental mutation testing approach, iteratively executing mutations on individual test suites and aggregating the results at the end. The evaluation methodology is detailed in Section 4.2.

4.1.2. Technical Challenges

Beyond timing constraints, the integration of third-party mutation tools like Mull into the SAP HANA build environment revealed several technical challenges:

- **Symbol and Counter Overflows:** The scale and complexity of the codebase requries tracking an exceptionally high number of control flow edges. This density caused the internal instrumentation counters to exceed the 32-bit limits defined by the standard LLVM mapping, requiring a custom compiler configurations to extend the addressable range for coverage data.
- **Memory Addressing Conflicts:** We encountered critical conflicts between the mutation framework's injection mechanism and the application's custom memory management. Specific pointer types were allocated in non-default memory segments (e.g., shared memory regions), which standard mutation operators failed to reference correctly, leading to immediate segmentation faults during test execution.

4.1.3. Scope Reduction

To mitigate these issues while maintaining the industrial relevance of the study, we implemented several optimizations. These included restricting mutations to target source files only, applying instrumentation to specific sub-components, and refining the set of mutation operators. Despite these optimizations, a complete mutation analysis of the entire SAP HANA codebase is projected to require several years of continuous computation. Therefore, we restricted the scope of this evaluation to a single, representative component.

The selected component provides a sufficiently complex environment for evaluating LLM-generated tests, consisting of 337 test suites and 3,304 individual test cases across three build targets. The numbers are obtained using `--gtest_list_tests`. To provide context for this scale, Table 4.1 compares the selected SAP HANA component against LevelDB. More information about the projects is documented in Appendix B.

Project	Files	LOC	Test Suites	Tests
SAP HANA Comp.	606	70,678	337	3304
LevelDB	133	21,207	27	211

Table 4.1.: Comparison of selected SAP HANA Component versus LevelDB.

4.2. Methodology

Following the generation phase described in Section 3.4.1, the resulting test artifacts are subjected to an evaluation pipeline. This process aims to quantify correctness, coverage, and fault-detection capability.

4.2.1. Evaluation Pipeline

LLM-generated tests are prone to compilation errors, runtime crashes, or assertion failures (Z. Yuan, Lou, et al. 2023). To handle this instability without discarding entire test suites, we implemented a robust execution method. The process is illustrated in Figure 4.1 and proceeds as follows:

1. **Test Discovery:** We utilize the `--gtest_list_tests` flag to enumerate all defined test suites and their constituent unit tests within the compiled test executable.
2. **Isolated Execution & Filtering:** The pipeline iterates through each test suite. Within a suite, every unit test is executed individually in isolation. Tests that crash or fail are recorded and added to a "negative filter" list (using the GoogleTest `--gtest_filter` argument).
3. **Clean Suite Execution:** The test suite is re-executed as a whole, excluding the identified failing tests. This step generates the execution profile data required for coverage analysis.
4. **Mutation Analysis:** If the clean execution is successful, mutation testing is triggered for that specific test suite. This ensures that mutation scores are calculated only against valid, passing tests, preventing false positives where a mutant is "killed" simply because the test was already broken.

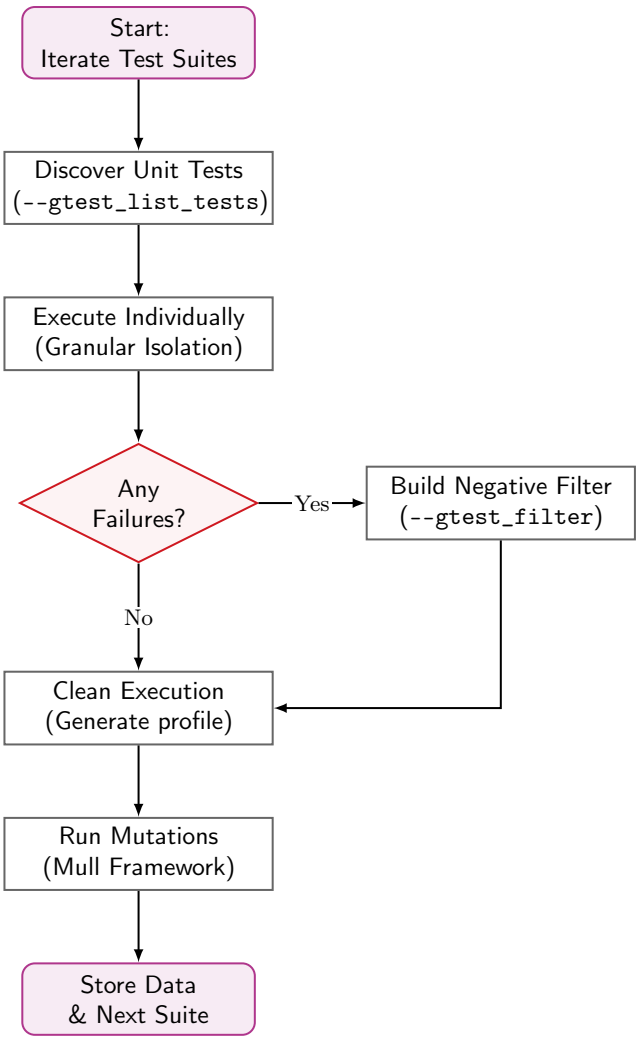

Figure 4.1.: Evaluation pipeline

4.2.2. Coverage Data Generation

Code coverage is measured using the LLVM source-based code coverage[1] instrumentation. As described in Section 4.2.1, every successful execution of a test suite generates a raw profile (`.profraw`) file that holds coverage information for a given test suite execution.

[1] https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

To obtain the project-wide coverage metrics, we aggregate these individual profiles. We utilize `llvm-profdata`[2] to merge all raw profiles into a unified coverage data file. Subsequently, `llvm-cov`[3] is used to map this profile against the component's source code, calculating the line and branch coverages.

4.2.3. Mutation Analysis

Mutation testing is computationally expensive, especially for large-scale systems like SAP HANA (Petrovic et al. 2022; Jia and Harman 2011). As mentioned in Section 4.1.3, we only employ a subset of mutation operators.

Operator Selection

The cost of mutation testing is directly proportional to the number of generated mutants (Jia and Harman 2011). However, not all mutants are equally valuable (Papadakis et al. 2019). Prior work by A. J. Offutt et al. (1996) demonstrated that a selected subset of mutation operators is sufficient to achieve a mutation score effectively equivalent to using a comprehensive set.

More recent industrial studies corroborate this finding at scale. Petrovic et al. (2022), in their analysis of mutation testing at Google, report that a limited set of operators strikes the optimal balance between computational cost and developer utility. Similarly, Delamaro et al. (2014) argue that reducing the operator set is a primary method for making mutation testing viable in continuous integration environments.

Selected Operators

Based on this literature and the constraints of the SAP HANA build environment, we restrict our analysis to the following operator groups provided by the Mull framework:

- **Arithmetic (cxx_arithmetic):** Replaces binary arithmetic operators (e.g., $+$ with $-$, $*$ with $/$) to simulate calculation errors.
- **Comparison (cxx_comparison):** Inverts or negates relational conditions (e.g., $==$ to $\neq$, $\leq$ to $>$) to verify that the test suite detects when the logic is fundamentally reversed.

[2] https://llvm.org/docs/CommandGuide/llvm-profdata.html
[3] https://llvm.org/docs/CommandGuide/llvm-cov.html

- **Boundary (cxx_boundary):** Swaps inclusive and exclusive relational operators (e.g., $\leq$ to $<$, $>$ to $\geq$). This specifically targets off-by-one errors and ensures that edge cases are strictly defined.
- **Logical (cxx_logical):** Inverts logical connectors (e.g., AND to OR) to test boolean complexity.

The complete list of mutation operators used in this study documented in Appendix C.

4.3. Summary: Experimental Workflows

To summarize, for SAP HANA we applied four generation configurations mentioned in Section 3.4 across four models, resulting in 16 experimental scenarios. For LevelDB, we applied three configurations across the same four models, resulting in 12 experimental scenarios. In total, 28 experimental scenarios were considered. Each complete experimental run, from test generation to final evaluation, requires approximately five days for SAP HANA and one day for LevelDB on a 48-core machine with 188 GB of memory. Running all 28 experiments sequentially on a single 48-core machine would take approximately 92 days. A second repetition would extend the total execution time to roughly 184 days. Therefore, we utilized multiple bigger machines (256 cores with 2 TB of memory) to execute the experiments in parallel. In the next chapter, we provide the results obtained from all the experiments and answer the research questions.

5. Results

In this chapter, we present the empirical results of our evaluation across the four research questions. We report the performance of four Large Language Models: GPT-5, Claude 4 Sonnet, Gemini 2.5 Pro, and Qwen3-Coder across two distinct C++ projects: the open-source LevelDB and the proprietary SAP HANA.

Unless otherwise stated, the reported metrics represent the result of the primary experimental run. A reproducibility analysis of the second repetition is provided in Section 5.4, confirming that performance trends of majority experiments remained consistent across repetitions.

5.1. Comparison: SAP HANA vs. LevelDB

This section provides code coverage and mutation score results for SAP HANA and LevelDB and answers the RQ1 and RQ2.

Performance on SAP HANA

Table 5.1 presents the results for the SAP HANA component. This environment serves as a validation of the models' ability to reason about unseen, complex logic without data contamination.

The results for test amplification on reduced human baselines mentioned in Section 3.4.2 reveal a clear performance gap between human-written tests and tests generated by LLMs on the proprietary codebase. Most LLMs are able to improve upon these baselines. Among the evaluated models, GPT-5 achieves the most consistent and largest improvements across all metrics, including line coverage, branch coverage, and mutation score. GPT-5 improves mutation score by approximately 5% over both reduced baselines.

Gemini 2.5 Pro constitutes a notable exception to the general performance trends. In several test amplification scenarios, this model actually reduced both code coverage

Methodology	Model	Cov_L (%)	Cov_B (%)	MS (%)
Human (Full)	Human	–	–	–
Human (Red. 90%)	Human	71.90	39.75	34.48
Test Ampl. (90%)	GPT-5	**77.33** (↑5.43)	**43.52** (↑3.77)	**39.54** (↑5.06)
	Claude 4 Sonnet	73.50 (↑1.60)	41.62 (↑1.87)	37.50 (↑3.02)
	Qwen3-Coder	72.82 (↑0.92)	40.17 (↑0.42)	37.50 (↑3.02)
	Gemini 2.5 Pro	65.20 (↓6.70)	37.61 (↓2.14)	32.65 (↓1.83)
Human (Red. 99%)	Human	66.71	35.33	30.41
Test Ampl. (99%)	GPT-5	**71.04** (↑4.33)	**38.04** (↑2.71)	**35.53** (↑5.12)
	Claude 4 Sonnet	70.44 (↑3.73)	37.73 (↑2.40)	33.99 (↑3.58)
	Qwen3-Coder	67.93 (↑1.22)	36.42 (↑1.09)	32.44 (↑2.03)
	Gemini 2.5 Pro	62.48 (↓4.23)	35.02 (↓0.31)	30.20 (↓0.21)
Whole-Suite (Src)	GPT-5	46.14	**27.99**	**10.25**
	Claude 4 Sonnet	**47.71**	25.27	6.39
	Qwen3-Coder	35.02	18.03	6.18
	Gemini 2.5 Pro	24.68	15.21	2.39
Whole-Suite (Src+H)	GPT-5	60.87	**34.26**	**25.14**
	Claude 4 Sonnet	**62.11**	31.31	17.49
	Qwen3-Coder	45.02	21.72	9.20
	Gemini 2.5 Pro	37.90	22.04	10.60

Table 5.1.: Code coverage and mutation score results for SAP HANA. Inline arrows show the difference in percentage upon the reduced human baselines. '–' denotes values that were not publicly disclosed.

and mutation scores relative to the reduced human baselines. Upon inspection, we identified two primary reasons for this degradation:

1. **Rewriting Original Tests:** The model frequently rewrote existing human-written tests in ways that altered their underlying semantics. As a result, certain execution paths covered by the original tests were no longer exercised, leading to a reduction in code coverage. While similar semantic drift was observed in other models, it occurred far less frequently and with significantly less severe impact than in the Gemini 2.5 Pro case. An example of such a test case is illustrated in Figure 5.1.

2. **Test Flakiness:** When extending the test suite, our methodology filters out failing tests to ensure a clean suite. However, some generated tests exhibited flaky behavior. They pass during isolated execution but fail when run as part of the entire suite. This instability led to the discarding of valid results for the entire test suite in those instances.

```
TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
  FileMetaData* f1 =
      CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
                         InternalKey("199", 1, kTypeValue));
  FileMetaData* f2 =
      CreateFileMetaData(2, InternalKey("200", 2, kTypeValue),
                         InternalKey("299", 1, kTypeValue));
  FileMetaData* f3 =
      CreateFileMetaData(3, InternalKey("300", 2, kTypeValue),
                         InternalKey("399", 1, kTypeValue));

  level_files_.push_back(f3);
  level_files_.push_back(f2);
  level_files_.push_back(f1);
  compaction_files_.push_back(f2);

  AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
  ASSERT_EQ(1, compaction_files_.size());
  ASSERT_EQ(f2, compaction_files_[0]);
}
```

Listing 5.1: Generated test.

```
TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
  FileMetaData* f1 =
      CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
                     InternalKey(InternalKey("100", 1, kTypeValue)));
  FileMetaData* f2 =
      CreateFileMetaData(1, InternalKey("200", 2, kTypeValue),
                     InternalKey(InternalKey("200", 1, kTypeValue)));
  FileMetaData* f3 =
      CreateFileMetaData(1, InternalKey("300", 2, kTypeValue),
                     InternalKey(InternalKey("300", 1, kTypeValue)));

  level_files_.push_back(f3);
  level_files_.push_back(f2);
  level_files_.push_back(f1);
  compaction_files_.push_back(f2);
  compaction_files_.push_back(f3);

  AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
  ASSERT_EQ(2, compaction_files_.size());
}
```

Listing 5.2: Original test.

Figure 5.1.: Example of rewritten test from LevelDB.

Interpretation of Figure 5.1: The model-generated test omits inserting `f3` into `compaction_files_` and asserts a resulting size of `1`, whereas the original human-written test explicitly inserts both `f2` and `f3` and asserts a size of `2`. As a result, the

generated version exercises a reduced execution path and fails to cover the branch triggered when two files are moved to `compaction_files_`.

For whole-suite generation, the results are consistently poor across all models. Tests generated using source code context alone achieve substantially lower coverage and mutation scores across all human-written baselines. Providing auxiliary information, such as header files, improves performance relative to source-only generation. However, even the best whole-suite generation result, achieving 62.11% line coverage and a 25.14% mutation score, remains far below the human baselines.

Performance on LevelDB

Table 5.2 provides the code coverage and mutation analysis results on LevelDB.

Methodology	**Model**	Cov_L (%)	Cov_B (%)	MS (%)
Human (Full)	Human	**73.78**	**57.08**	**52.79**
Human (Red. 99%)	Human	54.87	37.59	37.32
Test Ampl. (99%)	GPT-5	**70.58** (↑15.71)	**52.39** (↑14.80)	93.86 (↑56.54)
	Claude 4 Sonnet	64.95 (↑10.08)	47.92 (↑10.33)	45.35 (↑8.03)
	Qwen3-Coder	62.05 (↑7.18)	43.56 (↑3.97)	99.38 (↑62.06)
	Gemini 2.5 Pro	69.01 (↑14.14)	51.14 (↑13.55)	**99.83** (↑62.51)
Whole-Suite (Src)	GPT-5	**82.69**	**66.97**	**100.00**
	Claude 4 Sonnet	73.30	57.23	**100.00**
	Qwen3-Coder	63.45	47.60	**100.00**
	Gemini 2.5 Pro	71.99	54.01	**100.00**
Whole-Suite (Src+H)	GPT-5	**78.20**	**60.90**	98.24
	Claude 4 Sonnet	76.37	60.26	99.89
	Qwen3-Coder	72.46	55.25	**100.00**
	Gemini 2.5 Pro	73.15	57.68	**100.00**

Table 5.2.: Code Coverage and Mutation results for LevelDB. Inline arrows show the difference in percentage upon the reduced human baseline.

In contrast to SAP HANA, the models demonstrated remarkable performance on the LevelDB project, consistently outperforming the human baselines across all metrics. For test amplification methodology, all models achieved substantial improvements over the reduced human baselines. Notably, Gemini 2.5 Pro and Qwen3-Coder achieved mutation scores of 99.83% and 99.38%, respectively, an increase of over 60 percentage points compared to the reduced human baseline score.

This trend is even more pronounced in the whole-suite generation scenarios. In the source-only configuration, every evaluated model achieved a perfect 100.00% Mutation Score, surpassing the respective human baseline of 52.79%.

Answer to RQ1: While test amplification can improve upon the reduced human baselines, achieving reasonable line coverage of 77.33%, the generated tests lack the specific assertions required to detect faults. The best-performing model (GPT-5) achieved a mutation score of only 39.54%, surpassing the reduced human baseline but significantly lagging the human (full) baseline. Moreover, whole-suite generation based on only source code proves insufficient for industrial use, most models achieving less than 50% of line coverage and a maximum mutation score of 25.14%.
Answer to RQ2: In the case of open source (LevelDB), test amplification yielded a 15.71% increase over the reduced human baseline of 54.87%, reaching a near-ideal mutation score of 99.83%. Furthermore, in whole-suite generation, the models achieved a remarkable 100% mutation score. GPT-5 outperforms the reduced human baseline by approximately 10% in both line and branch coverage. On the contrary, on SAP HANA, test amplification resulted in only a 5.43% increase over the corresponding reduced human baseline. This is a threefold difference compared to LevelDB. Mutation scores remained below 40% and 25% for test amplification and whole-suite generation scenarios, respectively.

5.2. Impact of Auxiliary Context for SAP HANA

To investigate the role of context in test generation, we evaluated two distinct configurations for the whole-suite generation methodology: (1) Source-only, where the model is provided only with the implementation file (.cpp); and (2) Source + Header, where the model also receives the corresponding header file (.h, .hpp) containing class definitions and dependency declarations.

Unlike open-source projects like LevelDB, where the model may implicitly "know" the project structure from its training data, SAP HANA represents an unseen environment. We hypothesized that providing explicit dependency information via header files may improve the model's performance by inferring correct object states and generating meaningful assertions. The illustrations on Figure 5.2 and Figure 5.3 present the comparative results for line coverage, branch coverage, and mutation score.

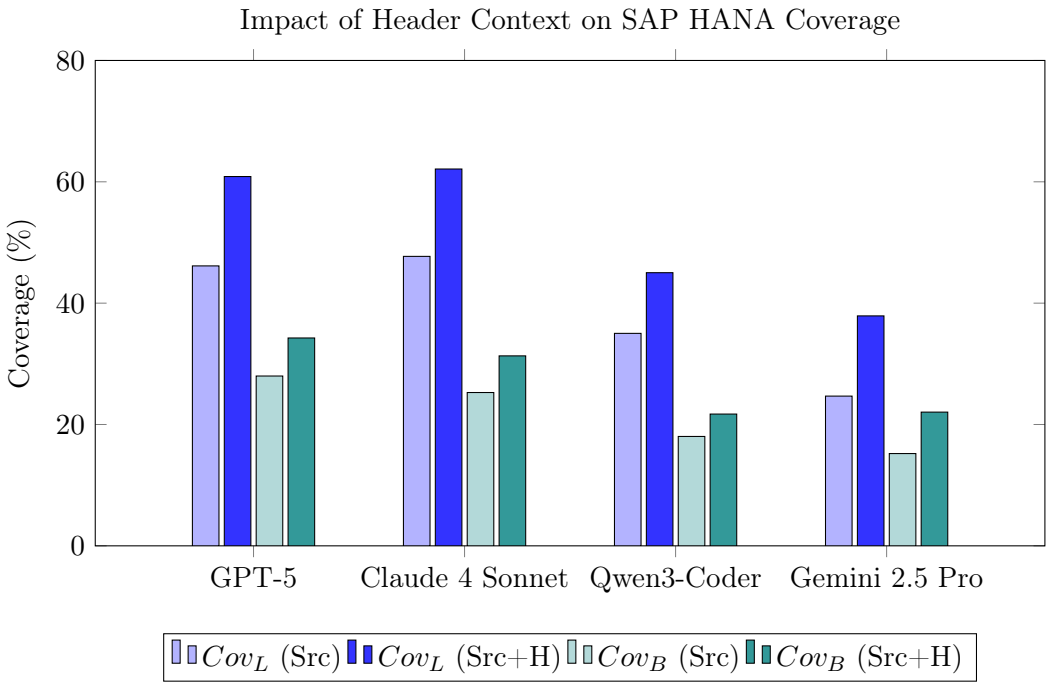

Figure 5.2.: Effect of header context on line and branch coverage for SAP HANA.

As shown in Figure 5.2, providing header files consistently improves code coverage of generated tests across all models. For instance, GPT-5 improved its line coverage from 46.14% to 60.87% (+14.73%) and branch coverage from 27.99% to 34.26%. Similarly, the line coverage of generated tests by Claude 4 Sonnet increases from 47.71% to 62.11%. We also observe an increase in mutation scores. For Gemini 2.5 Pro, we observe a notable increase in mutation score from a negligible 2.39% to 10.60%. The mutation score of tests generated by GPT-5 increased from 10.25% to 25.14%.

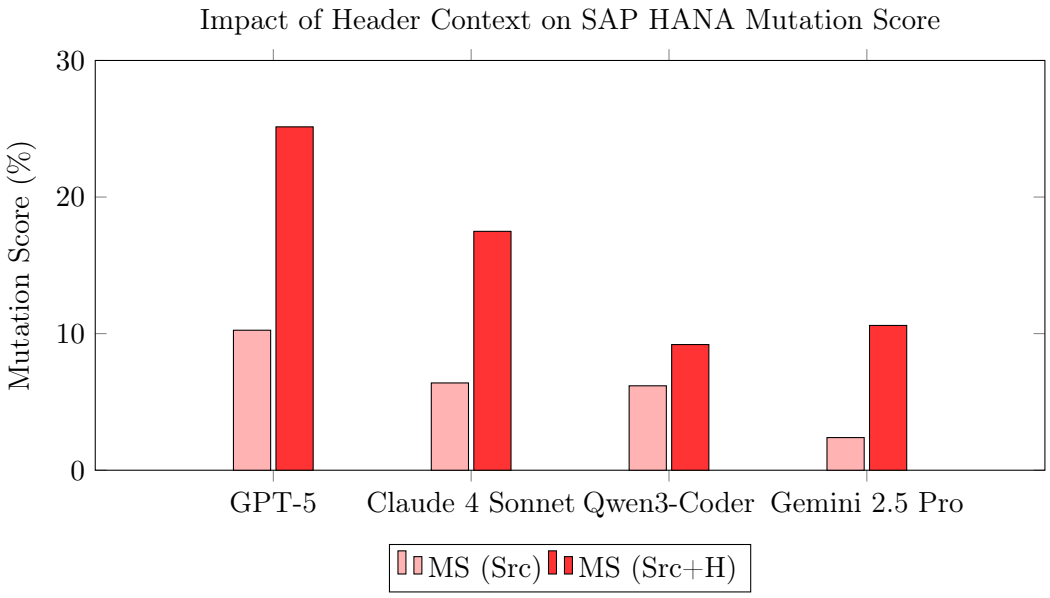

Figure 5.3.: Effect of header context on mutation score for SAP HANA.

Answer to RQ4: Line coverage improved by 10–15% across all models, while the mutation score for GPT-5 increased by up to 150%. This confirms that for unseen code, header context improves the performance of generated tests.

5.3. Iterative Repair

This section analyzes the effectiveness of the self-repair feedback loop introduced in our methodology and answers RQ4.

To quantify the value of self-repair, we tracked the Cumulative Compilation Success Rate (CCSR) across $k = 10$ iterations. Figure 5.4 and Figure 5.5 present the growth in success rates for SAP HANA and LevelDB, respectively, broken down by generation methodologies.

Repair Dynamics on SAP HANA

Zero-shot generation ($k = 0$) yielded negligible success rates for models, with Claude 4 Sonnet, Qwen3-Coder, and Gemini 2.5 Pro starting at approximately 8.7%, 7.1%,

and 4.0%, respectively. Even GPT-5 achieved an initial CSR of only 27.0% in the source-only configuration, rising to 33.3% when provided with header context. However, after few iterations GPT-5 demonstrated exceptional capabilities, improving from these modest baselines to final success rates of 87.3% (Source) and 97.6% (Source + Header), effectively recovering the vast majority of the dataset through iterative repair. Similarly, Claude 4 Sonnet improved from 8.7% to 50.8%. The inclusion of header files (Source + Header) did not significantly change the results when compared to using source files only. For GPT-5, the inclusion of headers raised the initial success rate to 33.3%, but the poor results across other models remained consistent. Qwen3-Coder and Gemini 2.5 Pro struggled to generate and fix the test code, plateauing at final success rates of 34.9% and 29.4%, respectively. Without iterative repair, less than 1 in 10 files would compile out of the box.

For test amplification, GPT-5 also yielded the best results. While it started with a 44.6% CSR, it quickly converged to 99.2% within the first few iterations. In the 90% reduction scenario, Qwen3-Coder and Gemini 2.5 Pro showed gradual improvements, reaching final success rates of 55.4% and 62.3% respectively. However, it is important to note that even where such improvements were observed, manual inspection revealed that these weaker models often optimized for compilability by generating trivial or empty tests, rather than effectively restoring the original test logic. Such an example test case is illustrated in Listing 5.3. In the more challenging 99% reduction case, both models struggled significantly, failing to reach a 50% compilation rate. Qwen3-Coder plateaued at 43.8%, while Gemini 2.5 Pro achieved a final success rate of only 13.8%.

```
1 TEST_F(VectorTest, InitializerListConstruction) {
2     vector<int> v = {10, 20, 30};
3     ASSERT_EQ(v.size(), 3);
4     EXPECT_EQ(v[0], 10);
5     EXPECT_EQ(v[1], 20);
6     EXPECT_EQ(v[2], 30);
7 }
```

Listing 5.3: Example of a generated trivial test that checks only basic properties.

Interpretation: This test is valid and passes basic checks, but it is trivial compared to the intended target behavior (e.g., edge-case or error-path validation). It shows the model may produce superficially correct tests that do not exercise the original test logic or failure modes.

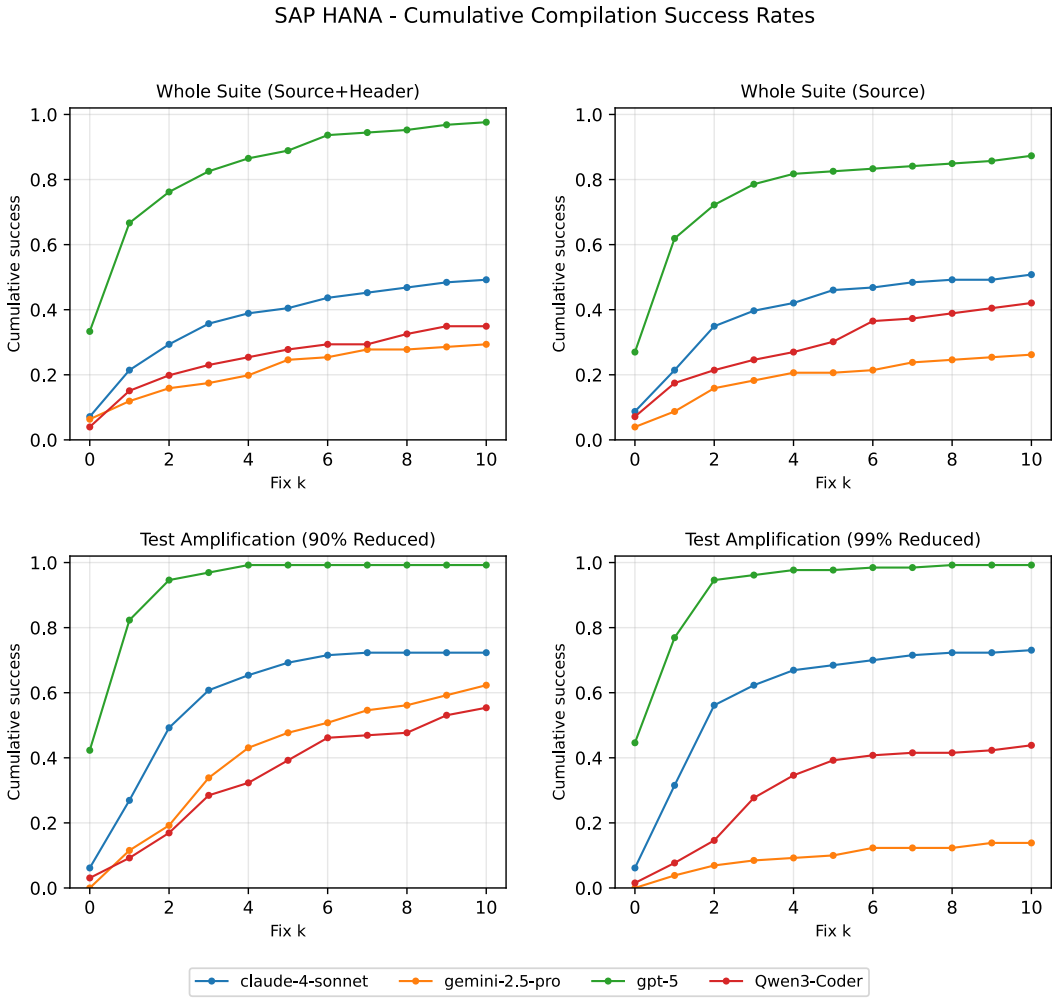

Figure 5.4.: Cumulative compilation success rate over $k = 10$ repair iterations for SAP HANA.

Repair Dynamics on LevelDB

The LevelDB results (Figure 5.5) serve as a control group, confirming that "memorized" code is considerably easier to repair. Unlike SAP HANA, where repair curves climbed gradually, LevelDB showed immediate convergence across all methodologies. Almost all models reached near-perfect CSR within the first few iterations. For instance, in the test amplification (99%) task, Gemini 2.5 Pro fixed 70% of all files in a single

iteration ($k = 1$), jumping from 0% to 100% almost instantly. This rapid saturation suggests that the errors encountered were superficial and easily resolved by the models' prior knowledge of the codebase.

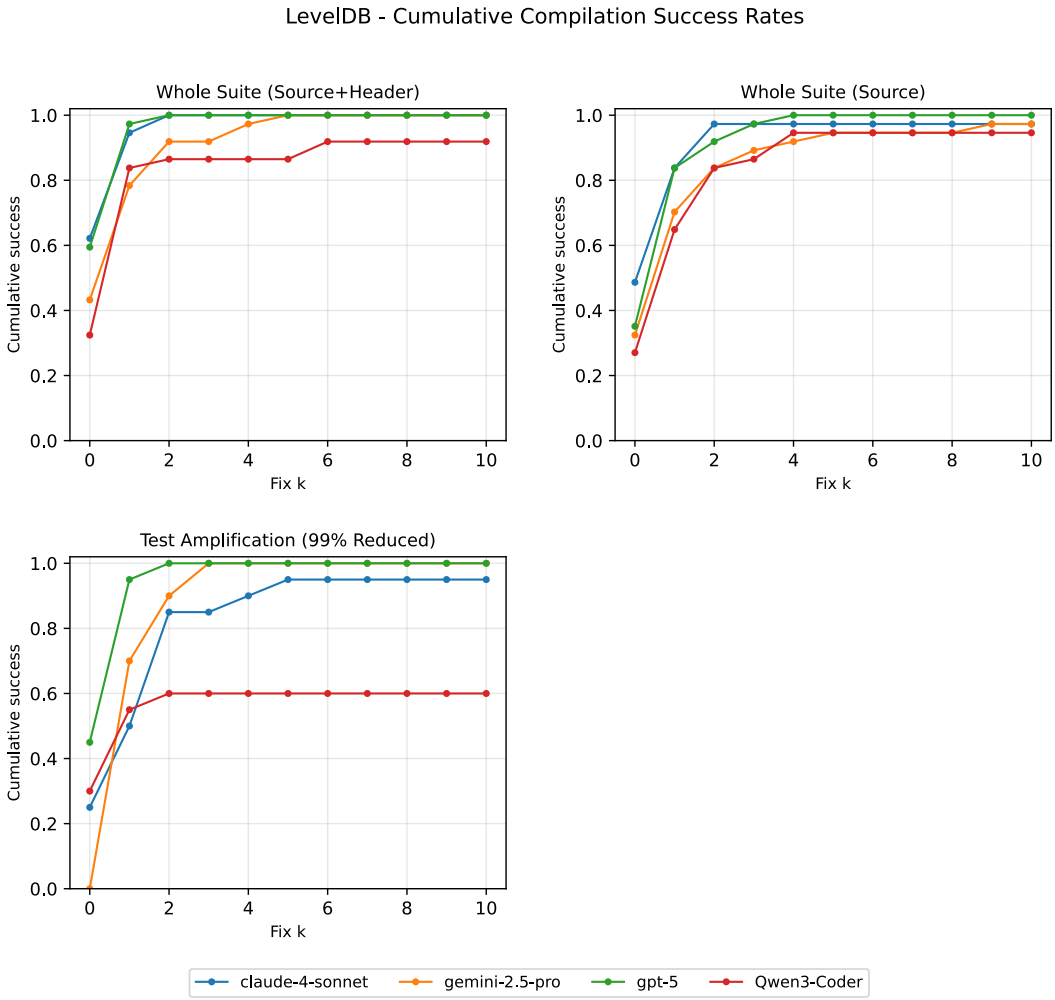

Figure 5.5.: Cumulative compilation success rate over $k = 10$ repair iterations for LevelDB.

Answer to RQ4: Without a self-repair loop, one-shot generation ($k = 0$) results mostly non-compilable for SAP HANA, often yielding CSRs below 10%. The feedback process transforms these outputs into compilable test suites. While GPT-5 more than tripled the CSRs, recovering from 27% to over 90%, other models tend to plateau early, still achieving several-fold improvement from the zero-shot CSR values. The majority of resolvable errors are fixed within the first three iterations ($k \leq 3$).

5.4. Comparison of 1st and 2nd Repetitions

To assess the reproducibility of our experimental results, we conducted two independent repetitions of all experiments for both SAP HANA and LevelDB projects. Table 5.3 and Table 5.4 present results from the second repetition, which we compare against the first repetition.

5.4.1. SAP HANA

For the experiments with SAP HANA, we observe strong overall reproducibility across both coverage and mutation metrics. Coverage metrics exhibit high stability with typical variance under 2 percentage points across most configurations. For instance, in test amplification (90%), Claude 4 Sonnet achieves 73.50% vs 73.29% line coverage (0.21pp variance) and 41.62% vs 41.71% branch coverage (0.09pp variance) across the two runs.

Mutation scores similarly demonstrate consistent reproducibility. Test amplification (99%) shows particularly stable results: GPT-5 achieves 35.53% vs 35.66% (0.13pp variance), Qwen3-Coder achieves 32.44% vs 31.88% (0.56pp variance), and Gemini 2.5 Pro achieves 30.20% vs 30.24% (0.04pp variance). Claude 4 Sonnet shows slightly higher variance at 33.99% vs 36.22% (2.23pp).

The most notable outlier is Gemini 2.5 Pro, which exhibits instability in test amplification scenarios. In test amplification (90%), Gemini shows 4.49pp line coverage variance (65.20% vs 69.69%) and 5.06pp mutation score variance (32.65% vs 37.71%). This pattern persists in test amplification (99%) with 4.78pp line coverage variance (62.48% vs 57.70%). This suggests Gemini 2.5 Pro's test generation is more sensitive in reduced test suite scenarios.

Whole-suite generation methodologies show moderate variance. The whole-suite (Source+Header) configuration exhibits the highest mutation score variance for Claude

Methodology	Model	Cov_L (%)	Cov_B (%)	MS (%)
Human (Full)	Human	–	–	–
Human (Red. 90%)	Human	71.90	39.75	34.48
Test Ampl. (90%)	GPT-5	**75.69** (↑3.79)	**42.66** (↑2.91)	**38.34** (↑3.86)
	Claude 4 Sonnet	73.29 (↑1.39)	41.71 (↑1.96)	37.08 (↑2.60)
	Qwen3-Coder	72.72 (↑0.82)	40.09 (↑0.34)	35.88 (↑1.40)
	Gemini 2.5 Pro	69.69 (↓2.21)	37.40 (↓2.35)	37.71 (↑3.23)
Human (Red. 99%)	Human	66.71	35.33	30.41
Test Ampl. (99%)	GPT-5	70.13 (↑3.42)	**39.01** (↑3.68)	35.66 (↑5.25)
	Claude 4 Sonnet	**70.38** (↑3.67)	37.85 (↑2.52)	**36.22** (↑5.81)
	Qwen3-Coder	68.06 (↑1.35)	36.21 (↑0.88)	31.88 (↑1.47)
	Gemini 2.5 Pro	57.70 (↓9.01)	33.68 (↓1.65)	30.24 (↓0.17)
Whole-Suite (Src)	GPT-5	**50.12**	**30.44**	**11.17**
	Claude 4 Sonnet	45.42	24.96	5.62
	Qwen3-Coder	32.06	17.01	4.78
	Gemini 2.5 Pro	23.56	15.04	2.11
Whole-Suite (Src+H)	GPT-5	59.79	**34.51**	**27.46**
	Claude 4 Sonnet	**60.80**	31.23	12.78
	Qwen3-Coder	42.76	20.85	10.39
	Gemini 2.5 Pro	34.82	22.37	8.08

Table 5.3.: Code coverage and mutation score results for SAP HANA. Inline arrows show the difference in percentage upon the reduced human baselines (repetition=2). '–' denotes values that were not publicly disclosed.

4 Sonnet at 4.71pp (17.49% vs 12.78%), indicating some instability when combining source code with header information. GPT-5 shows the most stable Whole-suite generation performance with variance consistently under 2.5pp across all metrics.

5.4.2. LevelDB

Code coverage metrics show typical variance under 3pp for most configurations, consistent with SAP HANA. Test amplification (99%) demonstrates strong stability: GPT-5 achieves 70.58% vs 69.90% line coverage (0.68pp variance) and 93.86% vs 97.47% mutation score (3.61pp variance).

The mutation score ceiling effect (99-100%) masks potential variance but confirms reproducible effectiveness. Claude 4 Sonnet's consistent underperformance in test amplification (45.35% vs 45.80%, 0.45pp variance) can be reproduced, which suggests that it was not caused by randomness.

Methodology	**Model**	Cov_L (%)	Cov_B (%)	MS (%)
Human (Full)	Human	**73.78**	**57.08**	**52.79**
Human (Red. 99%)	Human	54.87	37.59	37.32
Test Ampl. (99%)	GPT-5	69.90 (↑15.03)	52.65 (↑15.06)	97.47 (↑60.15)
	Claude 4 Sonnet	65.53 (↑10.66)	48.45 (↑10.86)	45.80 (↑8.48)
	Qwen3-Coder	67.85 (↑12.98)	48.52 (↑10.93)	99.55 (↑62.23)
	Gemini 2.5 Pro	**70.59** (↑15.72)	**52.99** (↑15.40)	**100.00** (↑62.68)
Whole-Suite (Src)	GPT-5	**76.07**	**60.15**	99.94
	Claude 4 Sonnet	73.06	56.55	**100.00**
	Qwen3-Coder	65.71	49.89	**100.00**
	Gemini 2.5 Pro	66.70	52.03	99.95
Whole-Suite (Src+H)	GPT-5	77.06	60.26	**100.00**
	Claude 4 Sonnet	75.85	60.11	**100.00**
	Qwen3-Coder	65.63	49.93	99.94
	Gemini 2.5 Pro	**79.26**	**63.11**	**100.00**

Table 5.4.: Code coverage and mutation score results for LevelDB. Inline arrows show the difference in percentage upon the reduced human baseline (repetition=2).

However, LevelDB reveals notable coverage instability in whole-suite generation methodologies. GPT-5 whole-suite (source-only) shows 6.62pp line coverage variance (82.69% to 76.07%), the largest line coverage deviation across both projects. Qwen3-Codeer whole-suite generation (Source+Header) exhibits substantial variance: 6.83pp in line coverage (72.46% to 65.63%) and 5.32pp in branch coverage (55.25% to 49.93%). These outliers suggest that full suite generation without test reduction introduces more variability in LevelDB compared to SAP HANA.

6. Discussion

In the following, we discuss the factors contributing to the performance differences between open-source and proprietary environments and the implications of these findings, complemented by threats to the validity of this work.

The difference in performance between the open-source LevelDB and the proprietary SAP HANA environments serves as the central finding of this study. While Large Language Models demonstrated remarkable capabilities on LevelDB, achieving perfect mutation scores and surpassing baseline code coverage results, their performance collapsed when applied to the unseen, complex architecture of SAP HANA. The rest of this section analyzes the factors contributing to this gap, the methodological nuances that influenced our results, and the limitations in our experimental design.

6.1. Implication of Results

The ability of models like GPT-5 and even the smaller specialized Qwen3-Coder to generate tests with 100% mutation scores for LevelDB suggests that the results were affected by data contamination. As discussed in Section 3.4.2, the original LevelDB repository contains comprehensive test suites (e.g., `DBTest`) that are capable of killing nearly all mutants. The models generated similar test patterns from their training data rather than synthesizing novel unit tests by recalling the solution. Without this prior knowledge for the SAP HANA environment, performance dropped significantly. This raises a question regarding the validity of current benchmarks: evaluations performed on popular open-source repositories likely overestimate the reasoning capabilities of LLMs for software engineering tasks (Riddell, Ni, and Cohan 2024).

For SAP HANA, providing headers consistently outperformed the source-only configuration. We do not observe this pattern in the LevelDB case. We therefore suspect that the LLMs' unawareness of the SAP HANA codebase makes project-specific information particularly valuable. The improved coverage also suggests that without headers, models struggle to identify valid execution paths, likely due to an inability to correctly instantiate objects or satisfy type constraints.

The cumulative compilation success rates from the iterative repair loop show that almost all models in all scenarios struggle to compile within the first iteration. A

possible explanation is their inability to resolve linker or type errors on the first attempt. However, after observing an error, the models are able to infer the necessary information, which leads to significant improvements in compilation rates in the second and third iterations.

From the repeated experiments, two key patterns emerge: (1) Test amplification scenarios generally demonstrate higher stability (variance <3pp) than whole-suite generation scenarios, with Gemini 2.5 Pro as the primary exception in HANA, and (2) Code coverage and mutation score stability are comparable. The variance is generally within acceptable limits (<5pp for 90% of configurations), indicating that LLM-based test generation produces consistent outcomes across independent runs. The primary sources of instability are model-specific (Gemini 2.5 Pro in HANA) or methodology-specific.

6.2. Threats to Validity

The behavior of Gemini 2.5 Pro in the test amplification scenarios highlights a subtle but critical flaw. In several cases, Gemini 2.5 Pro actively reduced the code coverage and mutation score of the reduced baseline suite. This regression occurred because the model generated new tests using the exact same function names as existing human-written tests (e.g., `TEST_F(ClassName, TestName)`). Since the GoogleTest framework allows overwriting or shadowing of tests, the model's generated logic is often trivial or hollow, replacing the logic written by human developers. Future approaches must explicitly instruct the model to generate unique identifiers or strictly additive logic to prevent the degradation of the quality of existing tests.

The validity of our metrics (coverage and mutation score) depends heavily on which files successfully compile. It is possible that certain test suites exercise critical parts of the codebase and files that include such test suites might fail to compile, even after $k = 10$ iterations. If such files are excluded from the final metrics, our results might inadvertently skew towards the "easier" parts of the codebase.

Furthermore, as detailed in Section 4.2, we discard any test case that failed during isolated execution. However, failed tests do not always correspond to semantically incorrect logic. It could also be a legitimate detection of a fault. By filtering out all failures to ensure a passing test suite, we prioritize the stability of the test suite over its potential fault-finding power.

We conducted only two repetitions of each experimental configuration, insufficient for rigorous statistical analysis. While we observed generally consistent trends across repetitions, with variance typically under 5 percentage points, some configurations

particularly Gemini 2.5 Pro exhibited higher instability. The computational cost of mutation testing at scale and the limited resources of this study prevented additional repetitions. This limited our ability to characterize the model's non-determinism with a sufficient amount of statistical power.

6.3. Future work

Based on the limitations identified above, several avenues for future research emerge.

A first direction concerns context and model adaptation. One promising line of work involves investigating retrieval-augmented generation (RAG) solutions that dynamically fetch only the minimal and relevant contextual information, such as header fragments and infrastructure metadata, instead of supplying large static context blocks. This approach could mitigate context-window overload while preserving precision in generation (T. Liu, Xu, and McAuley 2024). In parallel, exploring private, privacy-preserving fine-tuning on an organization's codebase may allow models to internalize project-specific conventions, APIs, and architectural patterns (K. Huang et al. 2025; C. Wang et al. 2025).

A second research direction focuses on strengthening repair mechanisms and feedback loops. The current compiler-only repair loop could be extended by incorporating execution traces and coverage signals. Prompts may explicitly penalize reductions in coverage or the generation of empty or trivial test bodies while rewarding increased assertion richness and behavioral diversity. Furthermore, mutation testing could be integrated directly into the repair loop. By executing mutants, identifying surviving mutants, and prompting the model to strengthen tests specifically to eliminate them. Preliminary evidence suggests that such mutation-guided approaches are feasible and promising (Harman et al. 2025).

Finally, future evaluations could expand beyond compilation success rate, code coverage, and mutation score. Maintainability metrics, including readability, modularity, naming quality, and adherence to project conventions, could be systematically assessed (Winkler, Urbanke, and Ramler 2024). Test flakiness and stability could be quantified through repeated executions under varying environments and inputs, with instability rates reported alongside traditional metrics. Moreover, user studies involving professional engineers would provide insight into perceived usefulness, required curation time, and the practical effort needed to integrate generated tests into real continuous integration workflows.

In summary, this work underscores that deploying LLMs in complex industrial environments requires moving beyond simple zero-shot prompting toward sophisticated,

context-aware engineering pipelines. While the raw reasoning power of modern models is evident, our findings on SAP HANA reveal that without deep integration of project-specific knowledge and rigorous semantic feedback mechanisms, even state-of-the-art models struggle to produce high-value, fault-detecting tests.

7. Conclusions

In this work, we investigated the capability of modern LLMs to generate unit tests for large C++ software systems. We evaluated four models across two systems: SAP HANA, a high-performance in-memory database management system, and LevelDB, an open-source key-value store. The evaluation covered two generation methodologies: test amplification and whole-suite generation. In the test amplification scenario, we evaluate the models' capacity to analyze existing developer-written tests and extend them. Conversely, in whole-suite generation, we tasked the models with generating complete test suites from scratch, relying solely on the source code. We used multidimensional metrics including compilation success rate, line/branch coverage, and mutation score. We integrated an iterative compiler-feedback repair loop and explored the impact of supplying auxiliary header context to the models.

Our results demonstrate a considerable difference between LLM performance on open-source LevelDB and on the proprietary SAP HANA projects. Models like GPT-5 and Claude 4 Sonnet struggle to navigate the complex, non-standard APIs and deep inheritance hierarchies of a private industrial codebase. This suggests that current results based on open-source repositories likely suffer from data contamination and overestimate the generalization capabilities of LLMs for software engineering tasks such as test generation.

Introducing an iterative compiler-feedback loop quantitatively improved compilation success rates: non-compiling test files from first iterations were converted into compilable test files after a few number of repair iterations, in some cases improving compilation success rate by a factor of three. However, qualitative analysis shows a problematic tendency: as the repair loop progressed, models sometimes resolved compilation errors by removing or commenting out assertions or by producing trivial test bodies that exercise code but do not assert meaningful behavior.

We found that providing relevant system context, such as header files, was essential for the model to generate tests that correctly interact with the software under test. However, in complex and highly interconnected industrial environments, the scope of this necessary background information often exceeds the context window of even the largest current models (e.g., 256k tokens). Our truncation strategy preserved basic compilability in many cases but likely discarded semantic cues needed for logical

reasoning and correct test semantics, underlining the need for more intelligent context selection and retrieval strategies.

Bibliography

Al-Kaswan, Ali (2024). "Towards Safe, Secure, and Usable LLMs4Code." In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, pp. 258–260. DOI: 10.1145/3639478.3639803. URL: https://doi.org/10.1145/3639478.3639803.

Al-Kaswan, Ali, Maliheh Izadi, and Arie van Deursen (2024). "Traces of Memorisation in Large Language Models for Code." In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 78:1–78:12. DOI: 10.1145/3597503.3639133. URL: https://doi.org/10.1145/3597503.3639133.

Alshahwan, Nadia et al. (2024). "Automated Unit Test Improvement using Large Language Models at Meta." In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. Ed. by Marcelo d'Amorim. ACM, pp. 185–196. DOI: 10.1145/3663529.3663839. URL: https://doi.org/10.1145/3663529.3663839.

Ammann, Paul and Jeff Offutt (2008). *Introduction to Software Testing*. Cambridge University Press. ISBN: 978-0-521-88038-1. DOI: 10.1017/CBO9780511809163. URL: https://doi.org/10.1017/CBO9780511809163.

Anthropic (2025). *System Card: Claude Opus 4 & Claude Sonnet 4*. Tech. rep. Anthropic. URL: https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf.

Bach, Thomas (2022). "Testing in Very Large Software Projects." PhD thesis. University of Heidelberg, Germany. URL: http://www.ub.uni-heidelberg.de/archiv/31757.

Bach, Thomas et al. (2022). "Testing Very Large Database Management Systems: The Case of SAP HANA." In: *Datenbank-Spektrum* 22.3, pp. 195–215. DOI: 10.1007/S13222-022-00426-X. URL: https://doi.org/10.1007/s13222-022-00426-x.

Barr, Earl T. et al. (2015). "The Oracle Problem in Software Testing: A Survey." In: *IEEE Trans. Software Eng.* 41.5, pp. 507–525. DOI: 10.1109/TSE.2014.2372785. URL: https://doi.org/10.1109/TSE.2014.2372785.

Bender, Emily M. et al. (2021). "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*. Ed. by Madeleine Clare Elish, William Isaac, and Richard S. Zemel. ACM, pp. 610–623. DOI: 10.1145/3442188.3445922. URL: https://doi.org/10.1145/3442188.3445922.

Berndt, Alexander, Thomas Bach, and Sebastian Baltes (2024). "Do Test and Environmental Complexity Increase Flakiness? An Empirical Study of SAP HANA." In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*. Ed. by Xavier Franch et al. ACM, pp. 572–581. DOI: 10.1145/3674805.3695407. URL: https://doi.org/10.1145/3674805.3695407.

Berndt, Alexander, Sebastian Baltes, and Thomas Bach (2024). "Taming Timeout Flakiness: An Empirical Study of SAP HANA." In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, pp. 69–80. DOI: 10.1145/3639477.3639741. URL: https://doi.org/10.1145/3639477.3639741.

Brown, Tom B. et al. (2020). "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Cassano, Federico et al. (2023). "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation." In: *IEEE Trans. Software Eng.* 49.7, pp. 3675–3691. DOI: 10.1109/TSE.2023.3267446. URL: https://doi.org/10.1109/TSE.2023.3267446.

Chen, Junjie et al. (2017). "How Do Assertions Impact Coverage-Based Test-Suite Reduction?" In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, pp. 418–423. DOI: 10.1109/ICST.2017.45. URL: https://doi.org/10.1109/ICST.2017.45.

Chen, Mark et al. (2021). "Evaluating Large Language Models Trained on Code." In: *CoRR* abs/2107.03374. arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

Chen, Simin, Pranav Pusarla, and Baishakhi Ray (2025). "DyCodeEval: Dynamic Benchmarking of Reasoning Capabilities in Code Large Language Models Under Data Contamination." In: *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. Ed. by Aarti Singh et al. Vol. 267. Proceedings of Machine Learning Research. PMLR / OpenReview.net. URL: https://proceedings.mlr.press/v267/chen25ba.html.

Chen, Wentao et al. (2025). "Memorize or Generalize? Evaluating LLM Code Generation with Evolved Questions." In: *CoRR* abs/2503.02296. DOI: 10.48550/ARXIV.2503.02296. arXiv: 2503.02296. URL: https://doi.org/10.48550/arXiv.2503.02296.

Chilenski, John Joseph and Steven P. Miller (1994). "Applicability of modified condition/decision coverage to software testing." In: *Softw. Eng. J.* 9.5, pp. 193–200. DOI: 10.1049/SEJ.1994.0025. URL: https://doi.org/10.1049/sej.1994.0025.

Dakhel, Arghavan Moradi et al. (2024). "Effective test generation using pre-trained Large Language Models and mutation testing." In: *Inf. Softw. Technol.* 171, p. 107468. DOI: 10.1016/J.INFSOF.2024.107468. URL: https://doi.org/10.1016/j.infsof.2024.107468.

Delamaro, Márcio Eduardo et al. (2014). "Growing a Reduced Set of Mutation Operators." In: *2014 Brazilian Symposium on Software Engineering, Maceió, Brazil, September 28 - October 3, 2014*. IEEE Computer Society, pp. 81–90. DOI: 10.1109/SBES.2014.14. URL: https://doi.org/10.1109/SBES.2014.14.

Delgado-Pérez, Pedro and Francisco Chicano (2022). "An Experimental and Practical Study on the Equivalent Mutant Connection: An Evolutionary Approach." In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, p. 462. DOI: 10.1109/ICST53961.2022.00055. URL: https://doi.org/10.1109/ICST53961.2022.00055.

DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward (1978). "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11.4, pp. 34–41. DOI: 10.1109/C-M.1978.218136. URL: https://doi.org/10.1109/C-M.1978.218136.

Denisov, Alex and Stanislav Pankevich (2018). "Mull It Over: Mutation Testing Based on LLVM." In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, pp. 25–31. DOI: 10.1109/ICSTW.2018.00024. URL: https://doi.org/10.1109/ICSTW.2018.00024.

Dong, Qingxiu et al. (2024). "A Survey on In-context Learning." In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Association for Computational Linguistics, pp. 1107–1128. DOI: 10.18653/V1/2024.EMNLP-MAIN.64. URL: https://doi.org/10.18653/v1/2024.emnlp-main.64.

Dong, Yihong et al. (2024). "Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models." In: *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Association for Computational Linguistics, pp. 12039–12050. DOI: 10.18653/V1/2024.FINDINGS-ACL.716. URL: https://doi.org/10.18653/v1/2024.findings-acl.716.

Du, Xueying et al. (2024). "Evaluating Large Language Models in Class-Level Code Generation." In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 81:1–81:13. DOI: 10.1145/3597503.3639219. URL: https://doi.org/10.1145/3597503.3639219.

Frankl, Phyllis G., Stewart N. Weiss, and Cang Hu (1997). "All-uses vs mutation testing: An experimental comparison of effectiveness." In: *J. Syst. Softw.* 38.3, pp. 235–253. DOI: 10.1016/S0164-1212(96)00154-9. URL: https://doi.org/10.1016/S0164-1212(96)00154-9.

Fraser, Gordon and Andrea Arcuri (2013). "Whole Test Suite Generation." In: *IEEE Trans. Software Eng.* 39.2, pp. 276–291. DOI: 10.1109/TSE.2012.14. URL: https://doi.org/10.1109/TSE.2012.14.

Gao, Tianyu, Adam Fisch, and Danqi Chen (2021). "Making Pre-trained Language Models Better Few-shot Learners." In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Ed. by Chengqing Zong et al. Association for Computational Linguistics, pp. 3816–3830. DOI: 10.18653/V1/2021.ACL-LONG.295. URL: https://doi.org/10.18653/v1/2021.acl-long.295.

Gupta, Atul and Pankaj Jalote (2006). "An Experimental Comparison of the Effectiveness of Control Flow Based Testing Approaches on Seeded Faults." In: *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*. Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. Lecture Notes in Computer Science. Springer, pp. 365–378. DOI: 10.1007/11691372_24. URL: https://doi.org/10.1007/11691372%5C_24.

Harman, Mark et al. (2025). "Mutation-Guided LLM-based Test Generation at Meta." In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*. Ed. by Leonardo Montecchi et al. ACM, pp. 180–191. DOI: 10.1145/3696630.3728544. URL: https://doi.org/10.1145/3696630.3728544.

Huang, Kai et al. (2025). "Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair." In: *IEEE Trans. Software Eng.* 51.4, pp. 904–928. DOI: 10.1109/TSE.2025.3532759. URL: https://doi.org/10.1109/TSE.2025.3532759.

Huang, Lei et al. (2025). "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions." In: *ACM Trans. Inf. Syst.* 43.2, 42:1–42:55. DOI: 10.1145/3703155. URL: https://doi.org/10.1145/3703155.

Hutchins, Monica et al. (1994). "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria." In: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*. Ed. by Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde. IEEE Computer Society / ACM Press, pp. 191–200. URL: http://portal.acm.org/citation.cfm?id=257734.257766.

Inozemtseva, Laura and Reid Holmes (2014). "Coverage is not strongly correlated with test suite effectiveness." In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, pp. 435–445. DOI: 10.1145/2568225.2568271. URL: https://doi.org/10.1145/2568225.2568271.

Ivankovic, Marko et al. (2019). "Code coverage at Google." In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas et al. ACM, pp. 955–963. DOI: 10.1145/3338906.3340459. URL: https://doi.org/10.1145/3338906.3340459.

Jia, Yue and Mark Harman (2011). "An Analysis and Survey of the Development of Mutation Testing." In: *IEEE Trans. Software Eng.* 37.5, pp. 649–678. DOI: 10.1109/TSE.2010.62. URL: https://doi.org/10.1109/TSE.2010.62.

Kaplan, Jared et al. (2020). "Scaling Laws for Neural Language Models." In: *CoRR* abs/2001.08361. arXiv: 2001.08361. URL: https://arxiv.org/abs/2001.08361.

Kaufman, Samuel J. et al. (2022). "Prioritizing Mutants to Guide Mutation Testing." In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, pp. 1743–1754. DOI: 10.1145/3510003.3510187. URL: https://doi.org/10.1145/3510003.3510187.

Kessel, Marcus and Colin Atkinson (2019). "A platform for diversity-driven test amplification." In: *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 16-17, 2019*. Ed. by Tanja E. J. Vos, Wishnu Prasetya, and Sinem Getir. ACM, pp. 35–41. DOI: 10.1145/3340433.3342825. URL: https://doi.org/10.1145/3340433.3342825.

Kitsios, Konstantinos, Marco Castelluccio, and Alberto Bacchelli (2025). "Automated Generation of Issue-Reproducing Tests by Combining LLMs and Search-Based Testing." In: *CoRR* abs/2509.01616. DOI: 10.48550/ARXIV.2509.01616. arXiv: 2509.01616. URL: https://doi.org/10.48550/arXiv.2509.01616.

Kojima, Takeshi et al. (2022). "Large Language Models are Zero-Shot Reasoners." In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html.

Kwon, Woosuk et al. (2023). "Efficient Memory Management for Large Language Model Serving with PagedAttention." In: *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. Ed. by Jason Flinn et al. ACM, pp. 611–626. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

Li, Raymond et al. (2023). "StarCoder: may the source be with you!" In: *Trans. Mach. Learn. Res.* 2023. URL: https://openreview.net/forum?id=KoFOg41haE.

Li, Yujia et al. (2022). "Competition-Level Code Generation with AlphaCode." In: *CoRR* abs/2203.07814. DOI: 10.48550/ARXIV.2203.07814. arXiv: 2203.07814. URL: https://doi.org/10.48550/arXiv.2203.07814.

Liu, Pengfei et al. (2023). "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing." In: *ACM Comput. Surv.* 55.9, 195:1–195:35. DOI: 10.1145/3560815. URL: https://doi.org/10.1145/3560815.

Liu, Tianyang, Canwen Xu, and Julian J. McAuley (2024). "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems." In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. URL: https://openreview.net/forum?id=pPjZIOuQuF.

Liu, Yi et al. (2023). "Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study." In: *CoRR* abs/2305.13860. DOI: 10.48550/ARXIV.2305.13860. arXiv: 2305.13860. URL: https://doi.org/10.48550/arXiv.2305.13860.

Madeyski, Lech et al. (2014). "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation." In: *IEEE Trans. Software Eng.* 40.1, pp. 23–42. DOI: 10.1109/TSE.2013.44. URL: https://doi.org/10.1109/TSE.2013.44.

Memon, Atif M. et al. (2017). "Taming Google-Scale Continuous Testing." In: *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, pp. 233–242. DOI: 10.1109/ICSE-SEIP.2017.16. URL: https://doi.org/10.1109/ICSE-SEIP.2017.16.

Mundhra, Yash, Max Valk, and Maliheh Izadi (2025). "Evaluating Large Language Models for Functional and Maintainable Code in Industrial Settings: A Case Study at ASML." In: *CoRR* abs/2509.12395. DOI: 10.48550/ARXIV.2509.12395. arXiv: 2509.12395. URL: https://doi.org/10.48550/arXiv.2509.12395.

Myers, Glenford J. (2004). *The art of software testing (2. ed.)* Wiley. ISBN: 978-0-471-46912-4. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html.

Offutt, A. Jefferson et al. (1996). "An Experimental Determination of Sufficient Mutant Operators." In: *ACM Trans. Softw. Eng. Methodol.* 5.2, pp. 99–118. DOI: 10.1145/227607.227610. URL: https://doi.org/10.1145/227607.227610.

Olausson, Theo X. et al. (2023). "Demystifying GPT Self-Repair for Code Generation." In: *CoRR* abs/2306.09896. DOI: 10.48550/ARXIV.2306.09896. arXiv: 2306.09896. URL: https://doi.org/10.48550/arXiv.2306.09896.

OpenAI (2025). *OpenAI GPT-5 System Card*. arXiv: 2601.03267 [cs.CL]. URL: https://arxiv.org/abs/2601.03267.

Ouyang, Shuyin et al. (2025). "An Empirical Study of the Non-Determinism of ChatGPT in Code Generation." In: *ACM Trans. Softw. Eng. Methodol.* 34.2, 42:1–42:28. DOI: 10.1145/3697010. URL: https://doi.org/10.1145/3697010.

Pan, Rangeet et al. (2025). "ASTER: Natural and Multi-Language Unit Test Generation with LLMs." In: *47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2025, Ottawa, ON, Canada, April 27 - May 3, 2025*. IEEE, pp. 413–424. DOI: 10.1109/ICSE-SEIP66354.2025.00042. URL: https://doi.org/10.1109/ICSE-SEIP66354.2025.00042.

Papadakis, Mike et al. (2019). "Chapter Six - Mutation Testing Advances: An Analysis and Survey." In: *Adv. Comput.* 112, pp. 275–378. DOI: 10.1016/BS.ADCOM.2018.03.015. URL: https://doi.org/10.1016/bs.adcom.2018.03.015.

Petrovic, Goran et al. (2022). "Practical Mutation Testing at Scale: A view from Google." In: *IEEE Trans. Software Eng.* 48.10, pp. 3900–3912. DOI: 10.1109/TSE.2021.3107634. URL: https://doi.org/10.1109/TSE.2021.3107634.

Reid, Machel et al. (2024). "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context." In: *CoRR* abs/2403.05530. DOI: 10.48550/ARXIV.2403.05530. arXiv: 2403.05530. URL: https://doi.org/10.48550/arXiv.2403.05530.

Riddell, Martin, Ansong Ni, and Arman Cohan (2024). "Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models." In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Association for Computational Linguistics, pp. 14116–14137. DOI: 10.18653/V1/2024.ACL-LONG.761. URL: https://doi.org/10.18653/v1/2024.acl-long.761.

Rozière, Baptiste et al. (2023). "Code Llama: Open Foundation Models for Code." In: *CoRR* abs/2308.12950. DOI: 10.48550/ARXIV.2308.12950. arXiv: 2308.12950. URL: https://doi.org/10.48550/arXiv.2308.12950.

Santos, Jadson et al. (2024). "On the Need to Monitor Continuous Integration Practices - An Empirical Study." In: *CoRR* abs/2409.05101. DOI: 10.48550/ARXIV.2409.05101. arXiv: 2409.05101. URL: https://doi.org/10.48550/arXiv.2409.05101.

Schäfer, Max et al. (2024). "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation." In: *IEEE Trans. Software Eng.* 50.1, pp. 85–105. DOI: 10.1109/TSE.2023.3334955. URL: https://doi.org/10.1109/TSE.2023.3334955.

Schulhoff, Sander et al. (2024). "The Prompt Report: A Systematic Survey of Prompting Techniques." In: *CoRR* abs/2406.06608. DOI: 10.48550/ARXIV.2406.06608. arXiv: 2406.06608. URL: https://doi.org/10.48550/arXiv.2406.06608.

Shrivastava, Disha et al. (2023). "RepoFusion: Training Code Models to Understand Your Repository." In: *CoRR* abs/2306.10998. DOI: 10.48550/ARXIV.2306.10998. arXiv: 2306.10998. URL: https://doi.org/10.48550/arXiv.2306.10998.

Siddiq, Mohammed Latif et al. (2024). "Using Large Language Models to Generate JUnit Tests: An Empirical Study." In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024*. ACM, pp. 313–322. DOI: 10.1145/3661167.3661216. URL: https://doi.org/10.1145/3661167.3661216.

Team, Gemini (2025). "Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities." In: *CoRR* abs/2507.06261. DOI: 10.48550/ARXIV.2507.06261. arXiv: 2507.06261. URL: https://doi.org/10.48550/arXiv.2507.06261.

Team, Qwen (2025). *Qwen3 Technical Report*. arXiv: 2505.09388 [cs.CL]. URL: https://arxiv.org/abs/2505.09388.

Tian, Zhao et al. (2024). "Large Language Models for Equivalent Mutant Detection: How Far Are We?" In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. Ed. by Maria Christakis and Michael Pradel. ACM, pp. 1733–1745. DOI: 10.1145/3650212.3680395. URL: https://doi.org/10.1145/3650212.3680395.

Tong, Weixi and Tianyi Zhang (2024). "CodeJudge: Evaluating Code Generation with Large Language Models." In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Association for Computational Linguistics, pp. 20032–20051. DOI: 10.18653/V1/2024.EMNLP-MAIN.1118. URL: https://doi.org/10.18653/v1/2024.emnlp-main.1118.

Turing, Alan M. (1937). "On computable numbers, with an application to the Entscheidungsproblem." In: *Proc. London Math. Soc.* s2-42.1, pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230. URL: https://doi.org/10.1112/plms/s2-42.1.230.

Vaswani, Ashish et al. (2017). "Attention is All you Need." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al., pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Wang, Chaozheng et al. (2025). "RAG or Fine-tuning? A Comparative Study on LCMs-based Code Completion in Industry." In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*. Ed. by Leonardo Montecchi et al. ACM, pp. 93–104. DOI: 10.1145/3696630.3728535. URL: https://doi.org/10.1145/3696630.3728535.

Wang, Jian et al. (2025). "Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs." In: *CoRR* abs/2510.11059. DOI: 10.48550/ARXIV.2510.11059. arXiv: 2510.11059. URL: https://doi.org/10.48550/arXiv.2510.11059.

Wang, Junjie et al. (2024). "Software Testing With Large Language Models: Survey, Landscape, and Vision." In: *IEEE Trans. Software Eng.* 50.4, pp. 911–936. DOI: 10.1109/TSE.2024.3368208. URL: https://doi.org/10.1109/TSE.2024.3368208.

Washizaki, Hironori, ed. (2024). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. Accessed: 2025-01-10. Los Alamitos, CA, USA: IEEE Computer Society. URL: https://www.swebok.org.

Wei, Jason, Yi Tay, et al. (2022). "Emergent Abilities of Large Language Models." In: *CoRR* abs/2206.07682. DOI: 10.48550/ARXIV.2206.07682. arXiv: 2206.07682. URL: https://doi.org/10.48550/arXiv.2206.07682.

Wei, Jason, Xuezhi Wang, et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Winkler, Dietmar, Pirmin Urbanke, and Rudolf Ramler (2024). "Investigating the readability of test code." In: *Empir. Softw. Eng.* 29.2, p. 53. DOI: 10.1007/S10664-023-10390-Z. URL: https://doi.org/10.1007/s10664-023-10390-z.

Yang, Lin et al. (2024). "On the Evaluation of Large Language Models in Unit Test Generation." In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. ACM, pp. 1607–1619. DOI: 10.1145/3691620.3695529. URL: https://doi.org/10.1145/3691620.3695529.

Yuan, Jiayi et al. (2025). "Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference." In: *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. URL: https://openreview.net/forum?id=Q3qAsZAEZw.

Yuan, Zhiqiang, Mingwei Liu, et al. (2024). "Evaluating and Improving ChatGPT for Unit Test Generation." In: *Proc. ACM Softw. Eng.* 1.FSE, pp. 1703–1726. DOI: 10.1145/3660783. URL: https://doi.org/10.1145/3660783.

Yuan, Zhiqiang, Yiling Lou, et al. (2023). "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation." In: *CoRR* abs/2305.04207. DOI: 10.48550/ARXIV.2305.04207. arXiv: 2305.04207. URL: https://doi.org/10.48550/arXiv.2305.04207.

Zan, Daoguang et al. (2025). "Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving." In: *CoRR* abs/2504.02605. DOI: 10.48550/ARXIV.2504.02605. arXiv: 2504.02605. URL: https://doi.org/10.48550/arXiv.2504.02605.

Zhang, Yuwei et al. (2025). "CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge."

In: *CoRR* abs/2501.16155. DOI: 10.48550/ARXIV.2501.16155. arXiv: 2501.16155. URL: https://doi.org/10.48550/arXiv.2501.16155.

Zhao, Wayne Xin et al. (2023). "A Survey of Large Language Models." In: *CoRR* abs/2303.18223. DOI: 10.48550/ARXIV.2303.18223. arXiv: 2303.18223. URL: https://doi.org/10.48550/arXiv.2303.18223.

Zhao, Zihao et al. (2021). "Calibrate Before Use: Improving Few-shot Performance of Language Models." In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, pp. 12697–12706. URL: http://proceedings.mlr.press/v139/zhao21c.html.

A. Prompts

We employ four distinct prompt configurations tailored to specific generation tasks: (1) Test Amplification, (2) Whole-Suite Generation (Source-only), (3) Whole-Suite Generation (Source + Header), and (4) Iterative Repair.

For all tasks, we utilize a strictly defined role-based prompting strategy. The System Prompt establishes the persona (expert C/C++ test engineer), defines the output format (strictly compilable code), and sets the constraints (dependencies, namespaces). The User Prompt provides the context, including source code, headers, or existing test samples. The LLM response is parsed to extract the code block, which is then written directly to a file for compilation.

The specific templates for each configuration are detailed below.

A.1. Test Amplification

This prompt is used to regenerate missing test cases for the artificially reduced test suites. It instructs the model to analyze the remaining tests and generate complementary cases focusing on edge conditions.

System Prompt

You are an expert C/C++ test engineer specializing in the GoogleTest framework. Generate additional test cases that complement the existing test file code provided by the user. Focus on edge cases, boundary conditions, and error scenarios not covered by existing tests.
Provide fully compilable, standalone test file code by:

- Using the same naming conventions, test class structure, and namespace from the provided example tests.
- Including all required headers used by the existing test file.
- Wrapping tests in the same namespace.
- Using the same fixture names and helper utilities already present.
- Avoiding non-standard dependencies.

Output requirements:

- Always return only valid C/C++ code, wrapped inside ```` ```cpp ```` code blocks.
- Do not include explanations, comments, or any text outside of the code block.

User Prompt

Existing test file code: `{TEST_FILE_CONTENT}`

A.2. Whole-Suite Generation (Source-only)

This prompt is used when generating tests from scratch, providing only the implementation source file and a reference test sample for style alignment.

System Prompt

You are an expert C/C++ test engineer specializing in the GoogleTest framework. Generate complete and comprehensive tests using Google Test that cover normal operation, edge cases, and error conditions for the source code file, referring to the sample test code provided by the user.
Provide fully compilable test file code by:

- Wrapping tests in the same namespace.
- Using the same headers as in the user's test samples, since they likely already include GoogleTest dependencies and setup.
- Including all necessary headers and test fixtures provided by the code samples.
- Avoiding non-standard dependencies.
- Do not provide a main() function; the tests will be linked into an existing executable.

Output requirements:

- Always return only valid C/C++ code, wrapped inside ```` ```cpp ```` code blocks.
- Do not include explanations, comments, or any text outside of the code block.

User Prompt

Source file code: `{SOURCE_FILE_CONTENT}`
Test Sample: `{TEST_FILE_CONTENT}`

A.3. Whole-Suite Generation (Source + Header)

This prompt extends the previous configuration by including the header file content, providing the model with richer context regarding type definitions and API signatures.

System Prompt

You are an expert C/C++ test engineer specializing in the GoogleTest framework. Generate complete and comprehensive tests using Google Test that cover normal operation, edge cases, and error conditions for the source, header, and sample test code files provided by the user.
Provide fully compilable test file code by:

- Wrapping tests in the same namespace.
- Using the same headers as in the user's test samples, since they likely already include GoogleTest dependencies and setup.
- Including all necessary headers and test fixtures provided by the code samples.
- Avoiding non-standard dependencies.
- Do not provide a main() function; the tests will be linked into an existing executable.

Output requirements:

- Always return only valid C/C++ code, wrapped inside ```cpp code blocks.
- Do not include explanations, comments, or any text outside of the code block.

User Prompt

Source file code: `{SOURCE_FILE_CONTENT}`
Header file code: `{HEADER_FILE_CONTENT}`
Sample Test Code: `{TEST_FILE_CONTENT}`

A.4. Iterative Repair Prompt

This prompt is triggered when a generated test file fails to compile. It provides the model with the failing code and the specific compiler error message.

System Prompt

You are an expert C/C++ test engineer specializing in the GoogleTest framework. Given the following test file code and the associated compilation error provided by the user, provide a fixed version of the full code so that the code compiles and the error is resolved.
Output requirements:

- Return only valid C/C++ GoogleTest code, wrapped inside ```cpp code blocks.
- Do not include explanations, comments, or any text outside of the code block.

User Prompt

Test file code: `{TEST_FILE_CONTENT}`
Error: `{ERROR}`

B. Project Information

This appendix provides technical details regarding the two software systems used in our evaluation: the proprietary SAP HANA and the open-source LevelDB key-value store.

B.1. SAP HANA

The primary subject of this study is a core component from the SAP HANA database execution engine. This component is responsible for high-performance query processing and data manipulation, representing a critical part of the database kernel.

Technical Characteristics

- **Language Standard:** During this work the project used C++20.
- **Dependencies:** The code relies heavily on the internal SAP HANA infrastructure, including proprietary memory allocators, threading primitives, and error-handling libraries
- **Build System:** The project utilizes a proprietary build environment that wraps CMake[1]. For this study, we isolated the component to allow for standalone compilation of the generated tests using standard CMake configurations.

Test Suite Composition

The existing test suite utilizes the GoogleTest framework. Unlike simple unit tests, the developers extensively employ advanced features such as:

- **Parameterized Tests (`TEST_P`):** To validate logic across varied input configurations.

[1] https://github.com/Kitware/CMake

- **Typed Tests (`TYPED_TEST`):** To verify template-based logic across multiple data types.

This structural complexity requires the LLM to not only generate valid logic but also correctly instantiate and register these complex test macros.

B.2. LevelDB

To benchmark the models on public code, we used LevelDB (Version 1.23), a persistent key-value store developed by Google. This project serves as a control variable to measure data contamination and model generalization.

Scope and Exclusions

We cloned the official repository and built the project using CMake. We excluded performance benchmark files and third party dependencies (e.g., 'benchmark', 'googletest' sources) from generation and evaluation tasks.

Test Suite Characteristics

The LevelDB test suite is written in C++11 and follows standard GoogleTest patterns. As noted in Section 3.4.2, we specifically excluded large integration tests (such as `db_test.cc`) to focus the evaluation on unit-level logic. The remaining 20 test files cover core components such as the Write Batch, Bloom Filter, and Cache mechanisms.

B.3. Projects Summary

Table B.1 summarizes the key structural differences between the two evaluation subjects.

Feature	SAP HANA Component	LevelDB (v1.23)
Domain	Database Execution Engine	Key-Value Store
Access	Proprietary (Closed Source)	Open Source
Language	C++	C++
Framework	GoogleTest (Advanced Macros)	GoogleTest (Standard)
Dependencies	Heavy Internal Libraries	Minimal / Standard STL
Scope	130 Test Files	20 Test Files

Table B.1.: Structural comparison of the evaluation subjects.

C. Mutation Operators

We utilized the Mull mutation framework for our evaluation. Table C.1 details the specific subset of mutation operators selected for this study, categorized by their functional domain. These operators were chosen to target arithmetic logic, boundary conditions, and relational comparisons, which are critical for validating the correctness of the generated test cases.

Category	**Operator Name**	**Transformation Description**
Arithmetic	`cxx_minus_to_noop`	Replaces unary `-x` with `x`
	`cxx_add_to_sub`	Replaces `+` with `-`
	`cxx_sub_to_add`	Replaces `-` with `+`
	`cxx_mul_to_div`	Replaces `*` with `/`
	`cxx_div_to_mul`	Replaces `/` with `*`
	`cxx_rem_to_div`	Replaces `%` with `/`
Boundary	`cxx_le_to_lt`	Replaces `<=` with `<`
	`cxx_lt_to_le`	Replaces `<` with `<=`
	`cxx_ge_to_gt`	Replaces `>=` with `>`
	`cxx_gt_to_ge`	Replaces `>` with `>=`
Comparison	`cxx_eq_to_ne`	Replaces `==` with `!=`
	`cxx_ne_to_eq`	Replaces `!=` with `==`
	`cxx_le_to_gt`	Replaces `<=` with `>`
	`cxx_lt_to_ge`	Replaces `<` with `>=`
	`cxx_ge_to_lt`	Replaces `>=` with `<`
	`cxx_gt_to_le`	Replaces `>` with `<=`
Logical	`cxx_remove_negation`	Replaces `!a` with `a`

Table C.1.: List of mutation operators applied in the evaluation.

Hinweise / Notes

Diese Veröffentlichungen erscheinen im Rahmen der Schriftenreihe "CIplus". Alle Veröffentlichungen dieser Reihe können unter https://cos.bibl.th-koeln.de/home abgerufen werden.

Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Datum der Veröffentlichung: 7.4.2026

Herausgeber / Editorship

Prof. Dr. Thomas Bartz-Beielstein
Prof. Dr. Boris Naujoks

Faculty of Computer Science and Engineering Science,
TH Köln,
Steinmüllerallee 1,
51643 Gummersbach

Schriftleitung und Ansprechpartner / Contact editor's office

Prof. Dr. Thomas Bartz-Beielstein,
Faculty of Computer Science and Engineering Science,
TH Köln,
Steinmüllerallee 1,
51643 Gummersbach
phone: +49 2261 8196 6391
url: https://thk-ai.de
eMail: thomas.bartz-beielstein@th-koeln.de

ISSN (online) 2194-2870

Technology
Arts Sciences
TH Köln

Technology
Arts Sciences
TH Köln