PALM: Synergizing Program Analysis and LLMs to Enhance Rust Unit Test Coverage

Bei Chu[†], Yang Feng[†*], Kui Liu[‡], Hange Shi[†], Zifan Nan[‡], Zhaoqiang Guo[‡] and Baowen Xu[†]

[†]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[‡]*Software Engineering Application Technology Lab, Huawei, Hangzhou, China*
{beichu, hangeshi, gzq}@smail.nju.edu.cn, {fengyang, bwxu}@nju.edu.cn, {kui.liu, nanzifan}@huawei.com

Abstract—Unit testing is essential for ensuring software reliability and correctness. Classic Search-Based Software Testing (SBST) methods and concolic execution-based approaches for generating unit tests often fail to achieve high coverage due to difficulties in handling complex program units, such as branching conditions and external dependencies. Recent work has increasingly utilized large language models (LLMs) to generate test cases, improving the quality of test generation by providing better context and correcting errors in the model's output. However, these methods rely on fixed prompts, resulting in relatively low compilation success rates and coverage.

This paper presents PALM, an approach that leverages large language models (LLMs) to enhance the generation of high-coverage unit tests. PALM performs program analysis to identify branching conditions within functions, which are then combined into path constraints. These constraints and relevant contextual information are used to construct prompts that guide the LLMs in generating unit tests. We implement the approach and evaluate it in 15 open-source Rust crates. Experimental results show that within just two or three hours, PALM can significantly improve test coverage compared to classic methods, with increases in overall project coverage exceeding 50% in some instances and its generated tests achieving an average coverage of 72.30%, comparable to human effort (70.94%), highlighting the potential of LLMs in automated test generation. We submitted 91 PALM-generated unit tests targeting new code. Of these submissions, 80 were accepted, 5 were rejected, and 6 remain pending review. The results demonstrate the effectiveness of integrating program analysis with AI and open new avenues for future research in automated software testing.

Index Terms—unit testing, automated test generation, large language models, Rust, program analysis, coverage.

I. INTRODUCTION

Software testing is crucial in the software development lifecycle, ensuring functionality and compliance with requirements [1], [2], [3]. Unit testing [4] is particularly effective in detecting defects early and maintaining code quality. Meanwhile, creating unit tests is often time-consuming, typically consuming over 15% of development time [5], [6]. Manually writing test cases is tedious and prone to errors, usually resulting in incomplete coverage and undetected bugs. This underscores the need for automated solutions to generate unit tests efficiently and accurately. Code coverage is the widely-used metric for evaluating the effectiveness of such automated tests. It measures the proportion of the source code (e.g., statements, branches) executed by the test suite, providing an objective indicator of its comprehensiveness. Achieving high code coverage is a primary goal in automated test generation, as it increases confidence in the software's correctness by exercising more of its behavior [7].

However, classic automated unit test generation methods aimed at achieving high coverage, such as search-based software testing (SBST) [8], [9], [10], [11], [12] and concolic execution [13], [14], [15], [16], have inherent limitations. SBST methods employ search algorithms to generate test inputs or call sequences, retaining those that execute correctly and enhance coverage and then mutate these inputs to maximize coverage. Despite automating test generation through search optimization, SBST suffers from high computational costs due to repeated program executions [9], a critical dependency on the fitness landscape which may prioritize coverage over fault detection [17], sensitivity to the curse of dimensionality in large input/state spaces [18], and difficulties in handling complex program semantics and external dependencies [19], limiting its practical efficacy. Meanwhile, concolic execution techniques, such as Syrust [13] and AUTS [20], can satisfy condition constraints and improve coverage by combining concrete and symbolic execution on test drivers. Nevertheless, they may suffer from path explosion [21], [22], [23], the computational complexity and scalability limitations of constraint solving [24], significant difficulties in accurately modeling environment interactions and external dependencies [25], [26], and the substantial engineering overhead required for developing and maintaining complex symbolic execution engines [27].

Recently, large language models (LLMs) have emerged as a promising solution to the limitations of classic test generation methods [28], [29], [30]. LLMs like OpenAI's GPT series [31] and DeepSeek [32] have demonstrated remarkable capabilities in understanding and generating natural language [33], [34]. Given that these LLMs are trained on enormous datasets, they can also understand and generate code [35], [36]. However, existing approaches leveraging LLMs for unit test generation often rely on fixed prompting strategies [37], [38], lacking explicit guidance on a method's internal structure and feasible execution paths. Consequently, when generating tests for methods with complex branch conditions, these approaches often fail to cover longer, intricate paths–a shortcoming that critically undermines the goal of achieving high coverage, a critical testing objective. Addressing this limitation necessitates techniques that explicitly decompose the problem based on the program's execution paths, enabling targeted guidance to the LLM for each distinct scenario.

*Yang Feng is the corresponding author.

Different from existing works, in this paper, we develop PALM, an approach that combines classic program analysis techniques with emerging large language models to generate unit tests satisfying specific path constraints. Recent research shows that breaking a problem into smaller steps and prompting LLMs to solve each step sequentially yields better results than directly asking for an answer [39], [40]. This enables LLMs to tackle more complex problems effectively. Based on this principle, we can decompose the task of generating unit tests into several smaller tasks, each corresponding to a specific execution path within the focal method. This decomposition allows LLMs to focus on specific subtasks, resulting in more accurate and diverse test cases. By combining these tests, we can create a comprehensive test suite that effectively covers various execution paths.

PALM begins with path analysis of the focal method to identify its execution paths, each representing a sequence of conditions and return values. We gather contextual information, including the focal method's body, other definitions, and function call details, to guide the LLM. For each path, we construct a prompt using this information and ask the LLM to generate test cases. We have implemented PALM for Rust and evaluated it on 15 open-source projects with 4145 focal methods. The GPT-4o mini-based prototype achieved an increase of over 20% in the generation of compilable tests and an improvement in coverage exceeding 50%. 91 unit tests, generated by PALM to cover code regions lacking prior developer-written tests, were submitted via pull requests. Subsequently, 80 of these tests were integrated into the project, representing an 94.12% acceptance rate among those reviewed.

In summary, this paper makes the following contributions:

- **Approach.** We introduce PALM, which combines program analysis with LLMs to generate high-coverage unit tests. This approach breaks the task into steps based on specific conditional paths within the focal method, using targeted prompts to guide the LLM.
- **Tool.** We implemented the PALM based on rustc APIs and designed a system that automatically generates unit tests through interaction with a large language model. The system is currently a prototype, with plans for open-sourcing upon completion.
- **Study.** We evaluated PALM on 15 open-source projects with 4145 focal methods. Our GPT-4o mini-based prototype increased the ratio of correctly generated, compilable tests by up to 20% and improved coverage by up to 50%.
- **Practical Usage.** 91 generated unit tests were submitted as pull requests to 8 distinct, popular open-source Rust projects. Project maintainers subsequently accepted and merged 80 of these submissions, accompanied by positive feedback. This represents an acceptance rate of 94.12% among the unit tests that underwent maintainer review.

II. BACKGROUND

A. Unit Test Generation

Unit test generation automates test creation for methods under test (focal methods), addressing the time-consuming and error-prone nature of manual testing [5], [6]. Classic approaches fall into two categories: search-based [12], [41], [8], [11] and symbolic execution-based [13], [15], [14], [42].

Search-Based Software Testing (SBST). Search-Based Software Testing (SBST) methods uses meta-heuristic algorithms (e.g., genetic algorithms) to iteratively mutate test cases for coverage improvement [18], [43]. Fundamentally, these tools operate by leveraging two core steps: (i) initially generating a random set of test cases as seed inputs and (ii) iteratively mutating these test cases to enhance the coverage of the focal method. Thanks to search algorithms, SBST methods can automatically explore possible test inputs and generate tests with higher coverage through optimized search techniques. However, SBST methods face the following challenges: (i) when dealing with complex and large systems, the search algorithms can consume substantial computational resources [44], (ii) they can get trapped in local optima [45], and (iii) the quality of the generated tests is influenced by the initial inputs and fitness landscape characteristics [46].

PALM decomposes the test generation task based on identified execution paths, each associated with specific branch condition constraints. This decomposition significantly reduces the search space that needs to be considered for each subtask, thereby reducing computational resource consumption compared to SBST. Additionally, by leveraging LLMs' ability to understand code semantics and constraints, PALM can generate tests that are more directly aligned with the requirements of a specific path, reducing the reliance on random initial inputs and their potential impact on result quality.

Concolic Execution. Concolic execution blends concrete and symbolic execution to collect path constraints and generate tests [25], [47], [48]. It uses a constraint solver to reason about path constraints, thereby generating test inputs. Through symbolic execution, this approach can systematically explore different execution paths in a program, generating high-coverage test cases. However, this method still faces several challenges in practice. Loops in the program can lead to path explosion, causing symbolic execution to consume substantial computational resources. Constraint solvers may also struggle with complex constraint conditions. Furthermore, symbolic execution often finds handling code involving external dependencies, such as file systems or networks challenging.

Similar to symbolic execution, PALM leverages path analysis to identify the constraints defining each execution path. However, instead of feeding these constraints to a classic solver, PALM uses the constraints as part of a natural language prompt to guide an LLM to generate concrete input values that satisfy the constraints for that specific path. This approach aims to bypass the computational complexity of constraint solving and the challenges in modeling external dependencies inherent in classic symbolic execution engines.

B. LLM for Code Generation

Large language models (LLMs) have demonstrated impressive capabilities in the field of natural language processing (NLP), excelling in various tasks such as conversation [34],

text generation [49], [50], [51], and reasoning [52]. Given their success in NLP, attention has naturally turned to code-related tasks, leading to the creation of numerous code generation models, such as AlphaCode [53], PolyCoder [54], Codex [36], Google's program synthesis model [55], and CodeGen [56].

The input text provided to the LLM during inference is called a prompt, which guides the model's generation process. Through prompt engineering, researchers can accomplish various code-related tasks without the need to retrain the model, such as code completion [57], code explanation [58], and code repair [59], [60], [61]. In this work, we apply prompt engineering techniques to construct structured prompts that explicitly incorporate the path constraints and context for each execution path, guiding the LLM to generate test cases.

III. DEFINITIONS AND MOTIVATING EXAMPLE

We present the definitions used throughout the paper and illustrate our objectives with an example.

Context. In this paper, the term *context* generally refers to the code related to, but distinct from, the focal method and the test functions. There are two types of context: (i) *focal context*, which is the context of the focal method, typically consisting of headers, macros, and global variables in the file containing the focal method; the declarations and definitions of the class to which the focal method belongs (if applicable); and the declarations and definitions of the data types and functions used by the focal method. (ii) *test context*, which is the context of the test functions, usually including the necessary use statements for the test files and any predefined data structures and variables required for the tests.

Condition Chain. By recording the branch conditions that must be satisfied for control flow transitions during path analysis, we can obtain approximate path constraints, which are sequences of conditions that must be met for a given execution path. In this paper, we refer to such sequences of conditions as condition chains.

```
1  fn wait(queue, mut curr_queue) {
2      loop {
3          if let Err(new_queue) = exchange {
4              if strict::addr(new_queue) & STATE_MASK !=
                    curr_state {
5                  return;
6              }
7              continue;
8          }
9          while !node.signaled.load(Ordering::Acquire) {
10             thread::park();
11         }
12         break;
13     }
14 }
```

Code 1: Motivating examples for `wait` function. Details of struct/trait definitions are omitted.

Next, we illustrate the challenges current test case generation methods face using the following example. Code 1 presents the `wait` method from the `once_cell` crate [62], which is used to block the current thread until a concurrent initialization completes. To better depict the method's control
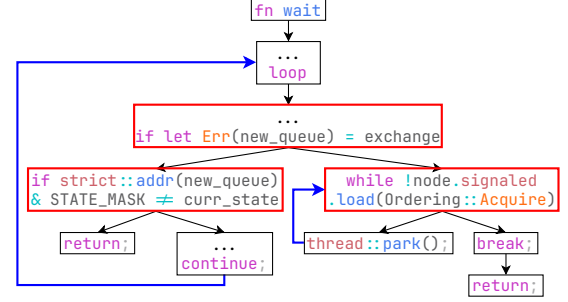

Fig. 1: Control flow graph of the focal method `wait`, highlighting the complex control flow involving conditional branches and nested loops that pose significant challenges for automated test generation.

flow behavior, we present its control flow graph (CFG) in Fig. 1. The CFG of the `wait` method reveals a structured composition of 2 conditional branches and 2 nested loops. To ensure comprehensive coverage, test inputs must be carefully crafted to meet the constraints.

Despite its power, symbolic execution struggles with functions like `wait`. Its nested loops can trigger a path explosion, overwhelming the analysis [26], while Rust's unique ownership model and concurrency primitives like AtomicPtr pose significant modeling challenges for symbolic engines, hindering scalability [63], [64]. Meanwhile, existing LLM-based approaches often fail because they (i) lack the specific context to correctly set up the function's preconditions [36], (ii) cannot generate the precise inputs needed to navigate the intricate control flow for high coverage [65], and (iii) frequently produce code that violates Rust's strict ownership and type safety rules, leading to low compilation rates. PALM is designed to overcome these specific challenges by synergizing program analysis with LLM generation.

IV. METHODOLOGY

In this section, we demonstrate the methodology for generating unit tests for a focal method, such as `wait`. PALM involves generating prompts corresponding to the execution paths within the focal method. This breaks down the task of generating a comprehensive test suite into several steps, each focused on generating tests for a single path. These prompts ultimately guide the model in generating unit tests with high coverage. When test cases fail to compile, we provide the model with erroneous code snippets and error messages, enabling it to suggest repairs iteratively.

A. Overview

The core of PALM involves collecting the chain of conditions along each path in the focal method through path analysis. By leveraging this technique, we can capture the program's behavior and gather the constraints along the execution paths.

Classic test generation methods based on symbolic execution enumerate all possible path constraints and generate test cases by solving these constraints [48]. However, real-world code often includes complex data structures and external
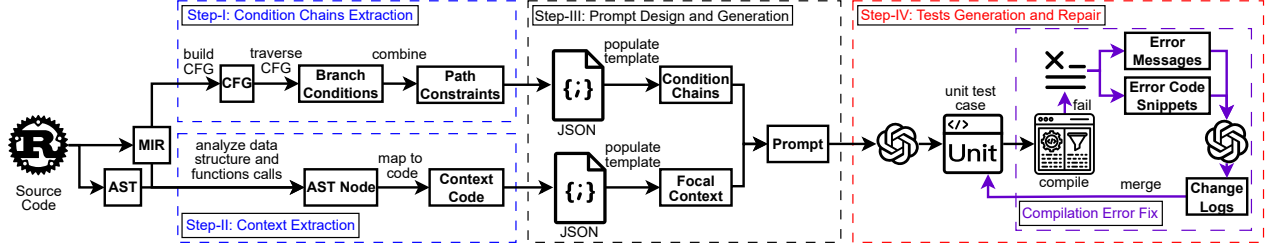
Fig. 2: The PALM framework. Step I extracts path constraints from the MIR-level CFG. Step II gathers contextual code via AST/IR analysis. Step III synthesizes a path-specific prompt. Step IV generates tests and uses iterative, compiler-guided repair.

dependencies, making constraint-solving challenging. Loops or deeply nested conditions can also lead to path explosion, consuming significant computational resources.

PALM addresses these challenges threefold. First, it retains complex constraints during condition chain collection, avoiding extensive solving. Second, it leverages the LLM's understanding of constraints and context for test case generation, instead of relying on constraint solvers. Finally, to reduce computational overhead, PALM selects a minimal path set covering all branch conditions, tackling this NP-hard set cover problem with a greedy approximation algorithm [66]. The number of generated tests is determined by the control flow complexity of the focal method. By default, PALM generates one test per minimized path. Additionally, it may generate one or two boundary tests for path conditions, budget permitting. The decision is based on the nature of the conditional expression: one test for single-boundary conditions (e.g., `x > 10`) and two for range-based conditions (e.g., `x >= 0 && x <= 10`). This strategy is designed to achieve full branch coverage efficiently while also probing critical edge cases.

Therefore, our test generation process consists of four steps: (I) collecting condition chains of the focal method; (II) gathering and constructing context for the focal method; (III) designing and populating a prompt template suitable for an LLM with the condition chains and context; and (IV) using the prompt to guide the LLM in generating unit tests and iteratively fixing any test cases that fail to compile. Fig. 2 provides an overview of PALM to test generation.

B. Step-I: Condition Chains Collection

We aim to generate tests that satisfy specific execution paths within the focal method. To achieve this, we must collect the relevant condition chains and provide this information to the model. This step involves a detailed explanation of how we traverse the focal method's Control Flow Graph (CFG) to gather the necessary condition chains.

1) Path Constraints Collection: Our approach leverages both the MIR and the HIR to derive conditional constraints from Rust source code. Unlike existing tools like MirChecker [67], which focus on flow-sensitive security analyses, our method is specifically designed to recover high-level semantics for unit test generation. MIR, a simplified version of Rust code, is typically used for flow-sensitive security analyses. It is constructed based on control flow graphs

(CFGs), where nodes represent basic blocks—sequences of instructions without branches or jumps—and edges represent control flow transitions, capturing potential execution paths and their associated conditional constraints. By analyzing these control flow transitions, we can extract the underlying conditional constraints. HIR is a compiler-friendly abstraction of the Abstract Syntax Tree (AST) and is closer to the syntactic structure of Rust source code compared to MIR.

Using the rustc API, we obtain the MIR and HIR of the focal method and perform a preorder traversal of the CFG to explore all possible execution paths. When encountering control flow transitions, we analyze the terminator of the basic block to extract and record the conditional constraints. Upon reaching the exit basic block, we record the return value, thereby constructing a conditional chain for the execution path.

Algorithm 1 Path Minimization

Require: set of path constraints P
Ensure: set of minimized path constraints M
    $M \leftarrow \emptyset$
    $S \leftarrow \text{getConstraints}(P)$
    while $S \neq \emptyset$ **do**
        $p_m \leftarrow \underset{p \in P}{\arg\max} |\text{getConstraints}(p) \cap S|$
        $M \leftarrow M \cup \{p_m\}$
        $S \leftarrow S \backslash \text{getConstraints}(p_m)$
    end while

---

2) Path Minimization: The number of possible execution paths in a method grows exponentially with the number of branches, making it impractical to generate test cases for all paths. To address this challenge, we aim to identify a minimal set of paths that can cover all branch conditions. Our approach decomposes the problem into two stages: path traversal and set cover. During path traversal, we collect all possible paths without filtering to ensure completeness. Subsequently, we formulate the problem of selecting a minimal set of paths as a classic minimum set cover problem.

Algorithm 1 details our approximate solution. It inputs a set of path constraints, each comprising multiple condition constraints, and outputs a minimal subset of these paths covering all condition constraints. The algorithm greedily constructs the minimized set M. It begins with a set S containing all unique branch conditions to be covered. In each iteration, it selects the path from the original set P that covers the maximum number

TABLE I: Condition Chains of `wait`

Path	Conditions				
	let Err(new_queue) = exchange	strict::addr(new_queue) & STATE_MASK != curr_state	let Err(new_queue) = exchange*	node.signaled.load (Ordering::Acquire)	node.signaled.load (Ordering::Acquire)*
0	false	—	—	true	—
1	false	—	—	false	true
2	true	true	—	—	—
3	true	false	false	true	—
4	true	false	false	false	true

*Loop-Terminating constraints.

of conditions currently in S. This path is added to M, and its corresponding conditions are removed from S. The process repeats until S is empty. By solving this problem, PALM significantly reduces the number of paths that need testing while ensuring complete branch coverage.

For example, consider the function `wait` shown in Code 1, after performing path analysis, we obtained the five condition chains illustrated in Table I. For the five paths in `wait`, we first select the path that covers the most conditions, path 4. Next, we choose path 0, which covers most of the remaining conditions, and finally, path 2. Thus, after minimization, we have selected three of the five paths marked red in Table I.

C. Step-II: Context Construction

Previous work on generating tests using LLMs has shown that incorporating additional context in the prompt enhances the quality of the generated tests [29]. To leverage this insight, we extract and incorporate relevant context through analysis. This context is categorized into two types:

- *Structure Context*: Definitions or declarations of member variables, methods, and structure fields associated with the focal method. This helps the model understand the focal method's role within its class or structure.
- *Dependency Context*: Declarations or definitions of parameter types, data types, and functions directly or indirectly used by the focal method. This ensures the generated code correctly handles external dependencies.

To manage token limits and reduce redundancy, we provide full definitions for directly used types and methods, using only relevant declarations for indirect dependencies.

The context extraction begins by parsing the focal file to extract use statements, global definitions, and type/function definitions, with use statements and global definitions being added to the context. If the focal method resides within a struct, the struct's definition and implementations (fields, associated functions, directly called methods) are extracted, while other methods receive only signatures. Definitions of direct dependencies–types and functions directly used by the focal method–are added. For indirect dependencies, local variable types and function calls are extracted by analyzing each function/method's Intermediate Representation (IR). Subsequently, a crate-wide Abstract Syntax Tree (AST) parse resolves use and mod dependencies, yielding a structured representation including trait bounds and return types. The final context aggregates file-level use/globals, direct/indirect dependencies (including struct/enum definitions, trait implementations), and information from call-graph/crate-wide analyses (e.g., trait bounds, return types). This assembled AST is then converted to Rust source code suitable for the LLM prompt.

D. Step-III: Prompts Design and Construction

Once the condition chains and context for the focal method are collected, we construct the prompt based on the focal method body, path constraints, return values, and other contextual information. To balance between overly simplistic prompts that might underestimate LLMs' capabilities and overly complex prompts that are impractical, we design our prompt by following established practices in unit test generation research [68], [38] and widely-recognized experiences in using ChatGPT [69]. Our prompt consists of two main components: (i) the requirement description part, which explains the task to LLMs, and (ii) the code context part, which includes the focal method and relevant context. An example for `wait` is shown in Fig. 3.

Code Context. Inspired by previous work [29], [68] on generating tests using LLMs and leveraging the content we have collected, the code context part includes the following components: (i) *focal context.* This is the context related to the focal method. (ii) *focal method.* This includes the signature and the body. (iii) *test context.* In some cases, the test context must also be included, as it comprises the necessary use statements and type declarations required for test execution.

Requirement Description. Based on established practices for using ChatGPT [68], our requirement description part includes (i) *role-playing instruction*, e.g., "as a software testing expert," to prime GPT for test generation [69], [70]. (ii) *task description instructions*, such as "please generate test functions based on the following guidelines:". To ensure the chat-tuned model generates only the required test cases without other descriptions, we include instructions like "provide the code in plain text format", and "without explanations." (iii) *condition chains*, to ensure that the model generates appropriate test cases. By structuring the prompt this way, we aim to maximize the model's ability to generate high-quality unit tests.

E. Step-IV: Test Generation and Repair

After constructing the initial prompt, we employ an iterative process to guide the model in generating test cases for each execution path. This decomposition by individual paths facilitates a comprehensive, high-coverage test suite. Fig. 4

```
As a software testing expert, please generate Rust test functions
based on the following guidelines:
1. Provide the code in plain text format, without explanations or Markdown.
2. ...
The context for the focal function is as follows:
// src/imp_std.rs
// crate name is once_cell
use std::{...};
const INCOMPLETE: usize = 0x0;
#[repr(align(4))]
struct Waiter {...};
fn wait(queue: &AtomicPtr<Waiter>, mut curr_queue: *mut Waiter) {...}
mod strict {
    #[must_use]
    #[inline]
    pub(crate) fn addr<T>(ptr: *mut T) -> usize
    where T: Sized, {
        unsafe { core::mem::transmute(ptr) }
    }
}
The function to be tested is presented as follows:
fn wait(queue: &AtomicPtr<Waiter>, mut curr_queue: *mut Waiter) {...}
Given the following constraints, potential panic-triggering statements,
and expected return values/types (all extracted from the function under test).
Generate test inputs that maximize the function's runtime satisfaction of
all constraints and expected outputs while considering panic conditions:
// constraint: exchange matches Err(new_queue) is true
// constraint: strict::addr(new_queue) & STATE_MASK ≠ curr_state is false
// constraint: exchange matches Err(new_queue) is false
// constraint: node.signaled.load(Ordering::Acquire) is false
// constraint: node.signaled.load(Ordering::Acquire) is true
```

Requirement Description

Focal Context / *Code Context*

Focal Method

Requirement Description / *Condition Chains*

Fig. 3: Structure of a PALM-generated prompt for the `wait` function. The Requirement Description part instructs the LLM and provides the target condition chain. The Code Context part supplies source code along with curated contextual definitions.

illustrates the test cases generated for `wait` using GPT-4o mini, demonstrating the effectiveness of our approach.

A significant challenge in generating test cases with LLMs is that the generated tests may fail to compile, compromising their effectiveness and coverage. Although truncating uncompilable tests generated by LLMs until they achieve successful compilation [71] might appear as a plausible solution, this practice often results in tests that are incomplete or even semantically invalid, consequently leading to diminished code coverage. A more robust solution is leveraging the model to fix compilation errors. Two established methods are: (i) providing the error information and context to the model, allowing it to regenerate the tests until they compile or a limit is reached [68]; and (ii) providing the error information and code snippets to the model while requesting a change log, which is then used to replace the erroneous code [61].

PALM adopts the second method for fixing compilation errors, as regenerating tests may inadvertently alter error-free parts. First, we extract error information and relevant code snippets from the compilation messages. For each error, we construct a prompt instructing the model to generate a change log based on this information. The code replacements are extracted from the model's response and applied before recompiling. This process iterates until the error is resolved or the iteration limit I is reached. To prevent the repair process from becoming excessively time-consuming, we also set an upper limit E on the number of errors to be fixed for each test function. In our experiments, we set E and I to 10 and 3, respectively. Our prompt design is inspired by prior work on using LLMs to fix Rust compilation errors [61], but we have streamlined the repair process to improve efficiency.

Fig. 5 illustrates fixing a compilation error in a test generated for the `wait` function. Initially, we compile the test and encounter a compilation error. We extract the error's location, line number, and message from the compiler output. In this example, the error occurs at line 4, with the message: error[E0369]: no implementation for `*mut imp::Waiter &

```
Test for path: !let Err(new_queue) = exchange -> node.signaled.load(Ordering::Acquire)
1 #[test]
2 fn test_01() {
3     let queue_value: *mut Waiter = INCOMPLETE_PTR;
4     let queue = AtomicPtr::new(queue_value);
5     let curr_queue: *mut Waiter = INCOMPLETE_PTR;
6     wait(&queue, curr_queue);
7     assert_eq!(queue.load(Ordering::Relaxed), curr_queue);
8 }

Test for path: let Err(new_queue) = exchange -> strict::addr(new_queue) & STATE_MASK != curr_state
1 #[test]
2 fn test_02() {
3     let queue = AtomicPtr::new(RUNNING as *mut Waiter);
4     let curr_queue = RUNNING as *mut Waiter;
5     wait(&queue, curr_queue);
6     let new_queue = queue.load(Ordering::Relaxed);
7     assert!(strict::addr(new_queue) & STATE_MASK == RUNNING);
8 }

Test for path: let Err(new_queue) = exchange -> strict::addr(new_queue) & STATE_MASK == curr_state
-> !let Err(new_queue) = exchange -> !node.signaled.load(Ordering::Acquire)
-> node.signaled.load(Ordering::Acquire)
1 #[test]
2 fn test_03() {
3     let queue = AtomicPtr::new(INCOMPLETE_PTR);
4     let curr_queue = INCOMPLETE_PTR;
5     let node = Waiter { ... };
6     let node_ptr = &node as *const Waiter as *mut Waiter;
7     queue.store(node_ptr, Ordering::Release);
8     wait(&queue, curr_queue);
9     assert_eq!(queue.load(Ordering::Acquire) as usize & STATE_MASK, COMPLETE);
10 }
```

Fig. 4: Example tests for `wait` generated by PALM, with one test shown per targeted execution path. Each test is crafted to satisfy a distinct sequence of predicates.

usize`. We then populate a prepared template with the error information and several lines of code surrounding the error. The model returns a corresponding change log based on the provided prompt format. From this change log, we identify the necessary code modifications. The correction involves adding the type conversion `as usize`, resolving the issue.

V. EXPERIMENT DESIGN, RESULTS AND DISCUSSION

We address the following RQs in our evaluation:

- **RQ 1. Effectiveness.** How effective is PALM in generating high-quality unit tests compared with baselines?
- **RQ 2. Ablation Study.** What is the impact of condition chains, context, and iterative compilation error repair on the effectiveness of generated unit tests?
- **RQ 3. Practical Usability.** How is PALM's usability for real world applications?

Experiment Settings. All our evaluations were conducted on a system equipped with an Intel Core i7-13700k processor, 64GB of RAM, running Ubuntu 24.04. The implementation of the path analysis and context extraction components of the PALM approach is based on Rust version nightly-2024-07-21. We validated PALM on GPT-4o mini [31], an advanced language model developed by OpenAI, designed for faster and more efficient natural language processing tasks.

Evaluation Metrics. Based on previous work in test generation [68], [29], [71], we have selected the following metrics:

1) **# Tests:** The number of generated tests for a project.
2) **Comp Pass:** The proportion of tests that can compile.
3) **Exec Pass:** The proportion of tests that successfully execute and pass.
4) **Line & Branch Coverage:** This metric reflects the effectiveness of the whole generated test suites. Line coverage is defined as the ratio of the number of lines of code executed to the total number of lines of code in the project after running all generated tests. Similarly, branch coverage is the ratio of the number of branches covered to the total number of branches.
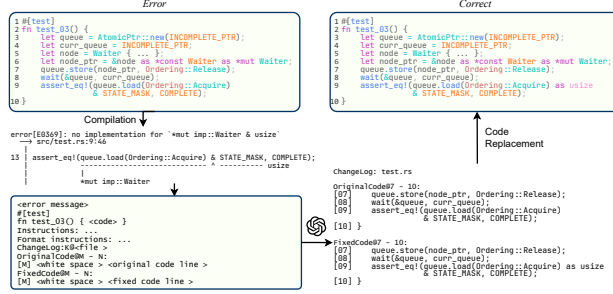
Fig. 5: The iterative repair process. 1) Capture the error message and surrounding code. 2) Prompt the LLM to generate a change log for a minimal fix. 3) The change is applied, and the code is recompiled. This loop continues until compilation succeeds or a predefined budget is exhausted.

Subject Programs. We selected 15 open-source Rust crates from crates.io [72] and GitHub to serve as our subject programs. These projects are chosen to form a challenging and representative benchmark, selected for their maturity, high popularity (most with over 300 million downloads), and domain diversity. They cover a wide range of applications, from data serialization and complex data structures to low-level systems programming and core algorithms, aiming to support the generalizability of our findings. In total, we selected 4145 focal methods to ensure a robust evaluation. Table II provides detailed information about these 15 crates, including their download counts, sizes, and other relevant characteristics.

Baselines. We selected five baselines to compare with PALM: two classic tools and three LLM-based approaches. The classic tools are RustyUnit [8], a search-based test suite generator for Rust, and RuMono [73], a fuzzing-based tool that extends RULF [74] with capabilities for generic APIs. The LLM-based baselines include: RUG [75], which lacks path-guided prompts and LLM-based repair, relying on fuzzing to enhance coverage; SymPrompt [71], which differs from PALM by using AST-based path analysis (instead of CFG-based) and truncating failing tests (instead of repairing them); and ChatTester [68], which contrasts with PALM's path-decomposed generation and patch-based repair by using one-shot generation and full regeneration for fixes.

A. RQ1: Effectiveness of PALM for Unit Test Generation

We evaluated PALM against five baseline tools on the subject crates detailed in Table II. For RuMono, we set a 24-hour timeout. We first measured the number of generated test artifacts (fuzz targets for RuMono; unit tests for others). We then assessed their compilation success rates. Successfully compiled tests were subsequently executed to determine their pass rates. Finally, we measured line and branch coverage for each project. Furthermore, we quantified the coverage achieved by the manually-authored unit tests within each selected project. Given the mature, popular, and extensively utilized nature of these projects, their existing test coverage levels were considered indicative of high-quality, manually-derived test suites achievable in practice.

TABLE II: Benchmark Crates

Crate	Description	Downloads	# Files	LOC
base64	Base64 decoding and encoding	567M	30	5k
bytes	Utility library for working with bytes	347M	33	6k
fastrand	Random number generator	302M	5	833
hashbrown	Rust port of SwissTable hash map	629M	35	11k
http	Library of common HTTP types	304M	34	10k
indexmap	Pure-Rust hash table	486M	31	9k
once_cell	One-time initialization for globals	439M	20	2k
qsv	Blazing-fast Data-Wrangling toolkit	225K	217	85k
quote	Turn Rust syntax tree into source code	582M	19	2k
rand	Random generators	516M	60	9k
regex	Search strings for matches of a regex	432M	222	97k
ryu	Pure Rust implementation of Ryū	418M	29	3k
serde	Framework for serializing and deserializing	517M	188	31k
serde_json	JSON serialization file format	451M	70	17k
strsim-rs	String similarity metrics	390M	3	1k

Results. Table III and IV present the comparative findings. While classic tools (RustyUnit, RuMono) achieve high compilation rates on simpler tests, PALM emerges as the clear leader among all LLM-based methods. It consistently achieves the highest compilation pass rate in this category, while maintaining a high and stable execution pass rate that is competitive with or superior to other LLM baselines across most projects. For instance, on hashbrown, PALM's compilation rate (41.19%) is over 6 times that of SymPrompt (6.63%). In terms of code coverage (Table IV), PALM's superiority is even more pronounced. It substantially surpasses all other baselines, both classic and LLM-based, across nearly every project. The performance gap is often dramatic; on http, PALM achieves 74.75% line coverage, more than 40 percentage points higher than the next best LLM-based tool, SymPrompt (29.07%). When benchmarked against these manually-authored tests, the original project tests exhibited an average line coverage of 70.94% and an average branch coverage of 66.47%. In comparison, tests generated by PALM surpassed the line coverage of the original tests in almost half of the evaluated projects, achieving an average of 72.30%. Notably, PALM-generated tests demonstrated superior branch coverage in the vast majority of projects, averaging 69.49%.

Discussion. The distinct test generation philosophies influence output characteristics: RustyUnit (SBST) produces comprehensive, longer unit tests, while LLM-based tools generate shorter tests, often targeting single input conditions. RuMono's fuzz target output scales with API density. We observed that, unlike the prolonged execution of RuMono, LLM-based tools typically complete test generation in 2-3 hours, varying with project scale and code complexity. A noteworthy observation is PALM's compilation pass rate, which is lower than that of classic tools like RustyUnit and RuMono. This is a direct consequence of PALM's ambition to generate more complex, human-like tests. Such tests often require intricate object setups and API call sequences, which can trigger the LLM's inherent limitations, such as factual inaccuracies or "hallucinations," making them more prone to compilation errors in a strict language like Rust [76]. This challenge motivates our inclusion of the iterative self-repair mechanism (Section IV-E). As shown in our ablation study (Table V), this mechanism is highly effective at boosting the final compilation rate and is essential for achieving high test coverage. While the compilation rate is lower than classic

TABLE III: Assessment of Generated Tests: Volume, Compilation, and Execution Success.

Crate	# Tests						Comp Pass (%)						Exec Pass (%)			
	RustyUnit	RuMono	RUG	SymPrompt	ChatTester	PALM	RustyUnit	RuMono	RUG	SymPrompt	ChatTester	PALM	RUG	SymPrompt	ChatTester	PALM
base64	50	16	228	187	266	629	**100.0**	**100.0**	46.05	27.81	40.60	48.38	72.38	57.69	**87.96**	77.78
bytes	47	300	—a	364	815	1189	51.06	57.67	—	54.40	51.41	**79.62**	—	82.86	81.25	**83.05**
fastrand	50	13	195	66	162	223	**100.0**	**100.0**	55.38	77.27	54.94	95.96	85.19	86.27	70.45	**90.34**
hashbrown	50	—	1391	347	1310	1527	**78.00**	—	30.70	6.63	34.73	41.19	**89.23**	69.57	88.13	82.54
http	—	213	1870	673	1322	2643	—	48.83	1.12	29.42	41.75	**53.35**	71.43	88.38	**91.85**	84.37
indexmap	48	—	1792	335	1619	1876	**100.0**	—	37.72	16.72	35.89	50.23	**90.38**	85.71	91.57	86.02
once_cell	42	8	7	103	211	299	40.48	75.00	28.57	45.54	43.13	**89.24**	50.00	**91.34**	84.62	90.94
qsv	47	—	1117	1989	1450	6483	**86.77**	—	45.30	27.75	52.76	52.43	67.59	63.77	**70.07**	65.49
quote	50	57	205	31	87	113	**100.0**	**100.0**	26.83	19.35	14.94	67.60	74.55	**100.0**	84.62	82.22
rand	50	—	1139	271	622	840	76.00	—	37.49	36.90	36.82	47.38	**84.31**	61.00	83.84	82.10
regex	50	130	3732	1004	3359	8251	94.00	**96.92**	37.73	27.39	40.19	35.48	83.59	72.91	84.59	**85.05**
ryu	50	6	90	115	188	512	94.00	66.67	**97.78**	77.39	85.64	87.73	52.27	59.55	65.84	**83.01**
serde	44	—	5356	954	1625	2274	**79.55**	—	1.72	5.97	19.57	13.32	82.61	89.47	89.94	**93.40**
serde_json	50	278	2449	955	2019	3116	48.00	**82.09**	19.44	15.81	22.44	44.19	81.09	85.74	85.98	**86.09**
strsim-rs	50	9	95	85	140	345	**100.0**	**100.0**	81.05	60.00	75.00	96.52	**92.21**	86.27	71.43	84.42

a The — symbol indicates that the tool was unable to generate tests for these projects.

tools, PALM generates a significantly larger volume of tests (see Table III). This ultimately results in a substantial pool of valid test cases, providing the breadth needed to achieve high coverage. Among LLM tools, PALM's superior compilation rates over others stem from its use of fine-grained contextual information and iterative self-repair of compilation errors.

Regarding coverage, classic SBST methods can be limited by initial inputs or local optima, and fuzzing often requires extensive runtime. LLMs, with their code understanding, can identify unique execution paths. PALM leverages path-specific requirements to guide generation. While this can sometimes be affected by conflicting constraints or model hallucinations [76], potentially explaining why improvements in coverage are not always substantial on every project. Nevertheless, this approach, particularly its CFG-based analysis and iterative repair mechanism, is the key driver of its overall success. In contrast, the AST-based analysis and test truncation strategy of SymPrompt proved less effective in the complex Rust environment. PALM's strategy of integrating static analysis for comprehensive function targeting, combined with its higher compilation success rates, allows it to generate tests for a broader set of functions within a project. This synergy of program analysis and effective compilation error correction enables PALM to achieve substantially higher coverage than alternatives, validating its approach and underscoring its potential for practical application. Across the 15 evaluated crates, PALM and the human-authored test suites each achieved superior line coverage in roughly half of the instances. Moreover, PALM demonstrated higher branch coverage in 10 out of the 15 crates, further underscoring its practical utility in software testing workflows. Instances where PALM did not attain superior line coverage were predominantly attributed to the extensive use of macros within those projects, which posed challenges for effective test generation. Conversely, PALM's consistent outperformance in branch coverage is likely due to its design focus on leveraging branch condition constraints to guide test generation, thereby often achieving more thorough exploration of conditional logic than typical manual efforts.

B. RQ2: Ablation Study

To optimize test generation, we integrated path constraint-based condition chains, focal context, and LLM-driven error resolution. We conducted an ablation study to evaluate the impact of these components, designing five prompt variants: (i) basic function body and task instructions; (ii) (i) + condition chains; (iii) (i) + context; (iv) (ii) + (iii); (v) (iv) + LLM error fixing, which is PALM. This systematic approach allowed us to quantify the individual contributions of each method.

Results. Table V details the ablation study results, revealing distinct impacts of different prompting strategies. Basic and context-only prompts generally produced fewer tests than condition chains-only prompts. Context-only prompts typically yielded higher compilation success rates (except on indexmap and strsim-rs) and better test pass rates (except on once_cell and rand). For code coverage, the general trend was context-only > condition chains-only > basic prompts, though basic prompts occasionally surpassed condition chains-only on some projects. Notably, prompts combining both context and condition chains generally achieved higher coverage than either component in isolation. The efficacy of model self-iteration for repair was evident when comparing configurations with and without this feature, showing universal improvements in compilation rates–exceeding 100% in some instances–which consequently boosted test pass rates and coverage. Ultimately, the full PALM methodology demonstrated superior performance, achieving the highest compilation rates and crucially, attaining the highest coverage across all evaluated projects.

Discussion. Our ablation study reveals that basic and context-only prompts, which direct the model to generate an entire test suite in one pass, typically produce fewer tests than condition chain-inclusive prompts designed to generate tests for specific execution paths. Context-only prompts, by incorporating the target method's external dependencies, enable the generation of tests that correctly utilize these dependencies, thus achieving higher compilation success rates. Although condition chain-driven prompts are theoretically poised for higher coverage, their increased propensity for compilation failures often leads to final coverage figures lower than those from context-only prompts. This finding is largely consistent with research on the SymPrompt technique for the GPT-4 model [71], suggesting that for dialogue-tuned models like GPT-4, contextual information contributes more significantly to effective test generation than explicit condition chains. Nevertheless, combining both context and condition chains generally yields further improvements across most projects. Moreover, leveraging LLMs to repair compilation errors sub-

TABLE IV: Evaluation of Test Suite Effectiveness via Code Coverage. Bold indicates the best result among all automated test generation tools. Manual coverage is provided as a human-authored reference benchmark.

Crate	Line Cov. (%)							Branch Cov. (%)						
	RustyUnit	RuMono	RUG	SymPrompt	ChatTester	PALM	Manual	RustyUnit	RuMono	RUG	SymPrompt	ChatTester	PALM	Manual
base64	6.36	15.67	**58.87**	43.33	21.65	58.14	93.49	10.58	17.42	58.17	41.18	19.08	**58.24**	87.50
bytes	27.53	9.01	—a	56.17	13.40	**83.93**	86.48	28.98	10.98	—	48.98	10.00	**75.14**	73.31
fastrand	58.59	35.02	60.61	61.28	18.86	**93.94**	63.64	62.50	7.50	66.96	52.63	18.42	**93.75**	49.11
hashbrown	7.16	—	47.50	7.75	3.31	**69.88**	53.40	5.22	—	54.21	8.00	4.86	**63.69**	60.57
http	—	33.94	11.85	29.07	1.77	**74.75**	64.38	—	31.75	11.31	29.03	2.84	**73.51**	59.35
indexmap	8.23	—	66.77	18.70	2.63	**74.30**	56.96	8.84	—	60.86	14.77	2.86	**69.47**	52.95
once_cell	42.96	11.93	1.68	50.84	8.11	**93.79**	80.43	49.34	11.54	1.68	43.82	7.87	**92.13**	75.00
qsv	9.81	—	28.94	38.43	14.36	**62.18**	78.83	8.74	—	28.26	26.65	9.89	**54.62**	69.49
quote	3.24	8.18	40.91	16.36	4.48	**77.27**	50.44	3.24	8.00	48.00	22.22	4.41	**70.34**	57.69
rand	14.13	—	**74.29**	42.59	5.76	61.83	78.06	14.44	—	**68.06**	36.97	5.37	54.62	66.52
regex	12.21	53.20	46.77	31.75	6.16	**55.49**	59.64	10.99	49.70	47.82	32.12	8.72	**55.61**	54.22
ryu	37.80	**98.27**	68.66	74.65	68.82	93.23	100.0	28.92	**98.04**	61.76	65.57	73.22	90.16	98.04
serde	8.11	—	1.35	14.38	4.90	**19.46**	57.00	9.39	—	1.61	13.59	6.81	**20.41**	55.46
serde_json	13.88	39.45	43.36	21.34	5.72	**66.65**	66.72	16.79	40.12	50.12	23.18	7.02	**73.28**	64.01
strsim-rs	64.72	82.78	95.00	80.52	63.61	**99.72**	74.17	50.00	81.82	88.64	71.43	54.55	**97.40**	73.86

a The — symbol indicates that the tool was unable to generate tests for these projects.

stantially boosts compilation rates on several projects, thereby markedly enhancing the quality of the generated tests.

C. RQ3: Practical Usability

To assess the practical usability of PALM-generated unit tests, we compared their achieved coverage against the existing test coverage in 15 projects, as detailed in Table IV. We then selected tests that demonstrably increased coverage and submitted them as PRs to these open-source projects.

Results. The outcomes of the submitted PRs are summarized in the final row of Table V, where '**P**' denotes PRs still pending developer feedback. In total, 91 tests were submitted across 8 projects. Of these, 80 were accepted and merged, 5 were rejected, and 6 are currently awaiting developer feedback.

Discussion. The target projects for this evaluation are popular, widely used, and actively maintained, possessing high preexisting test coverage that can be considered representative of diligent human development efforts. Despite this high baseline, PALM successfully generated tests that increased coverage in 8 of these projects. Among these, we contributed two PALM-generated unit tests to hashbrown, an officially maintained Rust crate that serves as the underlying implementation for HashMap and HashSet in the Rust standard library, thereby enhancing its test coverage and completeness. For projects where PALM did not identify coverage-enhancing tests, our investigation revealed extensive use of macros, a feature for which PALM currently lacks specialized handling, thus limiting its effectiveness in these specific contexts.

Developer interactions during the PR process indicated that while some expressed caution towards LLM-generated code, most were receptive, provided the tests were meaningful and demonstrably increased coverage. The 5 rejected tests were primarily due to developers seeking additional value beyond mere coverage improvement or perceiving the tests as redundant. Several tests were merged after incorporating developer feedback regarding code style, test descriptions, or usage patterns. Feedback is still pending for the remaining 6 tests. Overall, 87.91% of the submitted tests have been merged by developers. Considering only the tests that received developer review, the merge rate is 94.12%, underscoring the practical utility and acceptance of PALM-generated tests.

D. Qualitative Analysis of Failure Cases

While PALM demonstrates strong performance, its reliance on an LLM's heuristic understanding means it is not infallible. Failures typically arise when the LLM struggles to synthesize concrete, valid test code from the abstract path constraints provided by our analysis. This synthesis requires inferring a complete test scenario, including object setup and precise API call sequences. We identified two primary failure patterns: compilation failures due to misunderstanding language-specific semantics, and path constraint satisfaction failures due to a lack of causal reasoning.

```
1 pub fn choice<I>(iter: I) -> Option<I::Item>
2 where I: IntoIterator, I::IntoIter: ExactSizeIterator, {
3     with_rng(|r| r.choice(iter))
4 }
```

Code 2: Compilation failure example.

A common compilation failure mode involves the LLM applying programming patterns from other languages that violate Rust's strict ownership rules. For example, when testing the choice function in Code 2, which takes an iterator, PALM's LLM component might generate code that moves a Vec into the function and then attempts to reuse it, triggering a "use of moved value" error. The LLM fails because its general knowledge, trained heavily on languages like Python or Java where such reuse is common, overrides the specific constraints of Rust's move semantics.

```
1 fn reserve_inner(&mut self, additional, allocate) -> bool {
2     if kind == KIND_VEC {
3         unsafe {
4             let off = self.get_vec_pos();
5             if self.capacity() - self.len() + off >=
                   additional && off >= self.len() {
6             } else {
7                 if !allocate {
8                     return false;
9                 }
10             }
11             return true;
12         }
13     }
14 }
```

Code 3: Path constraint failure example.

TABLE V: Ablation Result of PALM

Method	Metric	base64	bytes	fastrand	hashbrown	http	indexmap	once_cell	qsv	quote	rand	regex	ryu	serde	serde_json	strsim-rs
Basic Prompt	# Tests	212	561	137	1073	1015	1219	185	1091	52	451	1199	126	1056	1438	109
	Comp Pass (%)	36.32	49.91	40.88	18.08	36.85	37.90	31.35	49.22	23.08	51.22	43.62	69.84	23.30	19.61	72.48
	Exec Pass (%)	96.10	83.93	83.93	77.60	91.98	95.02	89.66	74.10	100.0	85.28	89.29	57.95	93.90	87.27	88.61
	Line Cov. (%)	7.59	8.84	16.84	3.42	5.05	1.03	3.82	15.90	12.16	9.97	6.30	49.29	3.50	5.10	64.44
	Branch Cov. (%)	7.65	8.94	13.16	5.03	5.37	1.36	4.49	11.86	8.40	7.59		49.73	4.93	6.37	58.44
Condition Chains Only	# Tests	569	1042	207	1632	2311	1900	270	4936	85	769	2643	409	1873	2978	317
	Comp Pass (%)	30.23	41.84	41.06	13.66	28.26	29.53	23.33	43.35	12.94	42.78	35.34	73.84	16.60	15.28	82.62
	Exec Pass (%)	70.35	78.21	76.47	81.17	86.68	89.48	82.54	63.50	81.82	79.94	79.44	79.47	90.68	85.62	78.24
	Line Cov. (%)	31.09	18.35	15.83	2.66	10.93	2.76	0.72	25.17	15.95	11.38	7.09	58.58	3.60	4.81	68.61
	Branch Cov. (%)	32.94	12.57	14.06	4.18	11.35	4.30	1.12	18.49	16.21	9.66	9.03	54.10	5.42	6.92	62.34
Context Only	# Tests	216	647	146	990	1285	1187	209	1274	69	479	1307	160	1056	1564	117
	Comp Pass (%)	47.22	68.32	85.62	17.58	50.12	20.64	61.24	59.03	23.19	50.94	47.59	83.12	23.30	31.91	71.79
	Exec Pass (%)	84.31	83.94	93.94	81.61	88.82	90.61	95.31	73.01	87.50	68.85	81.03	69.92	93.90	89.65	85.71
	Line Cov. (%)	35.74	57.01	82.83	15.71	48.64	22.71	62.77	42.92	40.91	44.46	36.47	65.20	3.50	31.65	68.61
	Branch Cov. (%)	34.12	49.72	73.68	18.44	47.61	20.90	57.30	34.57	33.33	41.18	37.70	65.03	4.93	37.05	64.94
PALM w/o fix	# Tests	629	1189	223	1527	2643	1876	299	6483	113	840	3134	512	2274	3116	345
	Comp Pass (%)	40.38	59.71	90.58	19.19	45.48	25.16	56.52	50.96	25.66	47.02	40.62	73.44	13.28	26.70	86.96
	Exec Pass (%)	62.99	78.45	82.67	79.52	84.36	78.60	89.94	65.07	89.66	70.13	79.73	76.33	93.38	85.36	81.00
	Line Cov. (%)	57.28	63.79	91.89	17.92	60.72	33.01	68.02	61.47	44.45	47.30	43.88	74.33	16.54	31.28	89.44
	Branch Cov. (%)	58.24	58.94	92.19	22.63	57.95	30.53	66.29	54.25	37.04	45.38	43.06	73.77	15.27	35.86	84.42
PALM	# Tests	629	1189	223	1527	2643	1876	299	6483	113	840	3134	512	2274	3116	345
	Comp Pass (%)	48.38	79.62	95.96	41.19	53.35	50.23	89.24	67.60	52.43	47.38	54.22	87.73	13.32	44.19	96.52
	Exec Pass (%)	77.78	83.05	90.34	82.54	84.37	86.02	90.94	65.49	82.22	82.10	85.05	83.01	93.40	86.09	84.42
	Line Cov. (%)	58.14	83.93	93.94	69.88	74.75	74.30	93.79	62.18	77.27	61.83	55.49	93.23	19.46	66.65	99.72
	Branch Cov. (%)	58.24	75.14	93.75	63.69	73.51	69.47	92.13	54.62	70.34	54.62	55.61	90.16	20.41	73.28	97.40
	# Tests acc/rej	P(1)	12/0	19/0	2/0	—	39/0	3/0	—	—	5/5	—	—	—	—	P(5)

A more subtle failure occurs when a test compiles but does not trigger the intended path because the LLM fails to reason about the side effects of API calls. Consider satisfying the condition `off >= self.len()` inside the `reserve_inner` function in Code 3. Satisfying this condition requires more than just generating input values; it demands a sequence of state-manipulating actions. A correct test must first call a method like `self.advance()` to modify the object's internal state (`off`) before calling `reserve_inner`. The LLM often fails here because it cannot infer this non-local, causal dependency between an action (`advance()`) and its future consequence on an internal state check, highlighting a fundamental gap in reasoning about stateful object lifecycles.

E. Cost and Token Analysis

The practical adoption of LLM-based tools is intrinsically linked to their computational cost. To provide a clear picture of PALM's resource consumption, we analyzed its token usage and associated costs during our evaluation. On average, generating a single unit test required approximately 1500 input tokens and 300 output tokens. When the iterative repair mechanism was triggered, each repair round consumed an additional 800 tokens on average. For the entire evaluation across all 15 projects, which involved analyzing approximately 40,000 lines of code and generating around 65,000 test cases (including the full ablation study), the total consumption was approximately 800 million tokens. At the prevailing rates for the model used, this translated to a total cost of about $300. This analysis demonstrates that while not trivial, the cost is manageable for comprehensive academic studies and could be viable for integration into industrial pipelines, especially when applied selectively to critical or complex code modules.

VI. Discussion

A core design choice in PALM is to use an LLM as a heuristic constraint solver rather than a traditional SMT-based solver used in concolic execution. This choice embodies a fundamental trade-off between formal precision and practical applicability. We do not claim that LLMs are superior to dedicated solvers; for mathematical or logical constraints, the formal guarantees of SMT solvers are unparalleled. PALM, being heuristic, may fail on complex arithmetic constraints, a limitation we acknowledge. However, PALM's advantages emerge precisely where formal methods often falter: in handling the **semantic and environmental complexities** of real-world code. For instance, as shown in our motivating example (Section III), PALM successfully generates tests for functions involving concurrency primitives, which are notoriously difficult for concolic engines to handle without specialized, manually-crafted models. More broadly, LLMs excel at satisfying "semantic" constraints–such as correct API usage, complex object instantiation, and idiomatic patterns–by leveraging the vast knowledge in their training data. The high compilation success rates and positive developer feedback (Section V-C) from our study underscore this unique capability.

Furthermore, PALM offers crucial advantages in **scalability and engineering pragmatism**. It directly mitigates the path explosion problem, a key limitation of concolic execution, through its path minimization technique (Section IV-B2). By strategically selecting a diverse but minimal set of paths, it avoids the costly endeavor of exhaustive path exploration. This, combined with a significantly **lower engineering overhead** compared to developing and maintaining a symbolic execution engine, positions PALM as a more pragmatic and readily adaptable alternative. In essence, PALM trades formal precision on narrow constraints for superior handling of semantic complexity, greater scalability, and broader applicability in modern software systems.

VII. Threats to Validity

Threats from Model Selection. Our study relies exclusively on a single LLM, GPT-4o mini. This presents a threat to external validity, as our findings may not generalize to other models with different architectures or training data. We chose

GPT-4o mini for its strong performance and cost-effectiveness, but acknowledge that evaluating PALM with a diverse range of LLMs is a crucial next step. The core techniques of PALM, however, are model-agnostic and designed to be adaptable.

Subject Programs Selection. Our subject crates were selected from 15 popular and actively maintained projects on crates.io [72] and GitHub to cover diverse domains. This selection aims to enhance the applicability of our findings. However, we acknowledge that a study based on 15 projects, despite their diversity and complexity, may not fully represent the entire Rust ecosystem. Future work should expand this evaluation to an even larger and more varied set of crates.

Scope of Evaluation. Our evaluation primarily focuses on code coverage, a widely-used metric for test suite comprehensiveness. While high coverage is a prerequisite for effective bug finding, a large-scale study on bug detection effectiveness was beyond the scope of this work. We consider this an important direction for future research.

VIII. RELATED WORK

Search Based Software Testing (SBST). Randoop uses coverage-guided random test generation for Java and includes sanity-checking oracles to detect common bugs like NullPointerExceptions [77], [10]. EvoSuite is a framework that leverages SBST to generate tests for Java. It supports mutation testing and coverage-guided random test generation [9], [41], [78]. CITRUS is a C++ unit testing tool that generates high-coverage test suites by producing random sequences of method calls [12]. It is capable of handling challenging technical issues such as template instantiation. Pynguin utilizes SBST to generate regression tests for Python, aiming to ensure code reliability and detect regressions effectively [11]. RustyUnit introduces a search-based approach for automated unit test generation in Rust, utilizing a compiler wrapper and a many-objective genetic algorithm to achieve high code coverage [8]. PALM conceptually aligns with coverage-driven SBST methods, as both aim to generate test cases guided by coverage criteria. While SBST methods typically employ random search or optimization algorithms to identify suitable test inputs, PALM directs an LLM to generate inputs that satisfy constraints specified for a particular execution path.

Symbolic Execution based Test Generation. KLOVER is the first symbolic execution and automatic test generation tool for C++ [79], [15]. It is built on top of the symbolic execution engine KLEE [80] and extends it to support C++. Garg et al. propose a flexible testing framework that alternates between directed-random search and concolic execution to automatically cover hard-to-reach portions of the program [14]. Chen et al. propose a host-target split concolic testing method for embedded executables, performing symbolic execution on a resource-rich host and concrete execution on the target device, coordinated via cross-debugging, enabling testing without source code in the real environment [16]. SyRust leverages a novel logical encoding of Rust's ownership type system and polymorphic types to synthesize semantically valid test cases, employing symbolic execution and constraint solving to

effectively test Rust library APIs while ensuring type safety and proper API call chaining [13]. Unlike these approaches, we do not use classic constraint solvers to follow specific paths. Instead, we provide the source code of the focal method to the LLM, delegating the constraint-solving problem to the LLM, which then generates the tests.

LLM based Test Generation. Athenatest fine-tunes pre-trained transformers using paired method-test data and concludes that adding more context in prompts helps generate high-coverage tests [29]. Mathur et al. utilize T5 and GPT-3 to extract the conversation's context and topic, subsequently generating test cases [81]. Codamosa employs Pynguin, an SBST tool for Python, and iteratively invokes Codex to generate additional test cases for methods with low coverage [82]. Yuan et al. perform the first empirical study to evaluate ChatGPT's capability of unit test generation and propose ChatTester, which leverages ChatGPT itself to improve the quality of its generated tests [68]. MuTAP enhances LLM-generated tests through mutation testing and augmented prompts, demonstrating a significant increase in bug detection probability [83]. These methods rely on static prompts, whereas PALM combines program analysis with LLMs to generate different prompts for different paths, guiding the model to produce high-coverage test suites. RUG [75] introduces a semantic-aware bottom-up approach to leverage LLMs for generating unit tests in Rust, addressing compilation errors through context decomposition and enhancing test coverage by integrating coverage-guided fuzzing, achieving results comparable to human-written tests. SymPrompt [71] is the method most similar to PALM, as it generates corresponding prompts for different paths. However, unlike SymPrompt, we analyze the CFG instead of AST to explore the program's paths better. Meanwhile, instead of merely truncating the model's output, we leverage the model to fix compilation errors in the tests.

IX. CONCLUSION

This paper introduces PALM, which combines program analysis with large language models to generate high-coverage unit tests. PALM employs path analysis to obtain the condition chain for each path within the focal method. It then generates a prompt specific to each path by incorporating the context of the function. The model iteratively generates tests and, using provided error information suggests fixes for tests that fail to compile. Our experiments with GPT-4o mini demonstrate that PALM significantly improves both the compilation pass rate and coverage of tests generated by large language models.

References

[1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, no. 2002, 2002.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[3] S. Siddiqui, *Learning Test-Driven Development*. " O'Reilly Media, Inc.", 2021.

[4] K. Beck, *Test driven development: By example*. Addison-Wesley Professional, 2022.

[5] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[6] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[7] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[8] V. Tymofyeyev and G. Fraser, "Search-based test suite generation for rust," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 3–18.

[9] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[10] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[11] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[12] R. S. Herlim, Y. Kim, and M. Kim, "Citrus: Automated unit testing tool for real-world c++ programs," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 400–410.

[13] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of rust libraries with semantic-aware program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 899–913.

[14] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 132–141.

[15] H. Yoshida, G. Li, T. Kamiya, I. Ghosh, S. Rajan, S. Tokumoto, K. Munakata, and T. Uehara, "Klover: automatic test generation for c and c++ programs, using symbolic execution," *IEEE Software*, vol. 34, no. 5, pp. 30–37, 2017.

[16] T. Chen, X.-S. Zhang, X.-L. Ji, C. Zhu, Y. Bai, and Y. Wu, "Test generation for embedded executables via concolic execution in a real environment," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 284–296, 2014.

[17] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 205–214.

[18] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[19] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 224–234.

[20] L. Nguyen Tung, N. V. Binh Duong, K. N. Le, and P. N. Hung, "Automated test data generation and stubbing method for c/c++ embedded projects," *Automated Software Engineering*, vol. 31, no. 2, p. 52, 2024.

[21] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 351–366.

[22] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," in *2010 19th IEEE Asian Test Symposium*. IEEE, 2010, pp. 59–64.

[23] A. Kolchin, "A novel algorithm for attacking path explosion in model-based test generation for data flow coverage," in *2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC)*. IEEE, 2018, pp. 1–5.

[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[25] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[26] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[27] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[28] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24.

[29] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[30] A. Deljouyi, "Understandable test generation through capture/replay and llms," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 261–263.

[31] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[32] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[33] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, T. L. Scao, A. Raja *et al.*, "Multitask prompted training enables zero-shot task generalization," *arXiv preprint arXiv:2110.08207*, 2021.

[34] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[35] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[36] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[37] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Information and Software Technology*, vol. 176, p. 107565, 2024.

[38] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[39] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[40] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, "Star: Bootstrapping reasoning with reasoning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 476–15 488, 2022.

[41] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, 2013, pp. 362–369.

[42] S. Rho, P. Martens, S. Shin, and Y. Kim, "Taming the beast: Fully auto-mated unit testing with coyote c++," *arXiv preprint arXiv:2401.01073*, 2024.

[43] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[44] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *LASER Summer School on Software Engineering*. Springer, 2008, pp. 1–59.

[45] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[46] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–12.

[47] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[48] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[49] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[50] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[51] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[52] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[53] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[54] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[55] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[56] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, vol. 30, 2022.

[57] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[58] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, 2022, pp. 27–43.

[59] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[60] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[61] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, "Fixing rust compilation errors using llms," *arXiv preprint arXiv:2308.05177*, 2023.

[62] "GitHub - matklad/once_cell: Rust library for single assignment cells and lazy statics without macros — github.com," https://github.com/matklad/once_cell, [Accessed 30-04-2025].

[63] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of systems and software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[64] S. Bucur, "Improving scalability of symbolic execution for software with complex environment interfaces," Ph.D. dissertation, EPFL, 2015.

[65] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[66] N. Alon, B. Awerbuch, and Y. Azar, "The online set cover problem," in *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, 2003, pp. 100–105.

[67] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 2183–2196.

[68] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[69] J. Kemper, "Chatgpt guide: Use these prompt strategies to maximize your results," https://the-decoder.com/chatgpt-guide-prompt-strategies/, 2024, (Accessed on 03/07/2025).

[70] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.

[71] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[72] "Crates.io: Rust package registry," https://crates.io/, [Accessed 30-04-2025].

[73] Y. Zhang, J. Wu, and H. Xu, "Rumono: Fuzz driver synthesis for rust generic apis," *ACM Transactions on Software Engineering and Methodology*, 2024.

[74] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 581–592.

[75] X. Cheng, F. Sang, Y. Zhai, X. Zhang, and T. Kim, " RUG: Turbo LLM for Rust Unit Test Generation ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 634–634. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00097

[76] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[77] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.

[78] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, pp. 611–639, 2015.

[79] G. Li, I. Ghosh, and S. P. Rajan, "Klover: A symbolic execution and automatic test generation tool for c++ programs," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 609–615.

[80] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[81] A. Mathur, S. Pradhan, P. Soni, D. Patel, and R. Regunathan, "Automated test case generation using t5 and gpt-3," in *2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS)*, vol. 1. IEEE, 2023, pp. 1986–1992.

[82] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[83] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024.