Evaluating and Improving Automated Repository-Level Rust Issue Resolution with LLM-based Agents

Jiahong Xiang[†]
Research Institute of Trustworthy
Autonomous Systems, Southern
University of Science and Technology
Shenzhen, China
xiangjh2022@mail.sustech.edu.cn

Wenxiao He
Southern University of Science and
Technology
Shenzhen, China
12110818@mail.sustech.edu.cn

Xihua Wang
Southern University of Science and
Technology
Shenzhen, China
12213006@mail.sustech.edu.cn

Hongliang Tian
Ant Group
Hangzhou, China
tate.thl@antgroup.com

Yuqun Zhang[†*]
Research Institute of Trustworthy
Autonomous Systems, Southern
University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

Abstract

The Rust programming language presents a steep learning curve and significant coding challenges, making the automation of issue resolution essential for its broader adoption. Recently, LLM-powered code agents have shown remarkable success in resolving complex software engineering tasks, yet their application to Rust has been limited by the absence of a large-scale, repository-level benchmark. To bridge this gap, we introduce Rust-SWE-bench, a benchmark comprising 500 real-world, repository-level software engineering tasks from 34 diverse and popular Rust repositories. We then perform a comprehensive study on Rust-SWE-bench with four representative agents and four state-of-the-art LLMs to establish a foundational understanding of their capabilities and limitations in the Rust ecosystem.

Our extensive study reveals that while ReAct-style agents are promising, i.e., resolving up to 21.2% of issues, they are limited by two primary challenges: comprehending repository-wide code structure and complying with Rust's strict type and trait semantics. We also find that issue reproduction is rather critical for task resolution. Inspired by these findings, we propose RustForger, a novel agentic approach that integrates an automated test environment setup with a Rust metaprogramming-driven dynamic tracing strategy to facilitate reliable issue reproduction and dynamic analysis. The evaluation shows that RustForger using Claude-Sonnet-3.7 significantly outperforms all baselines, resolving 28.6% of tasks on Rust-SWE-bench, i.e., a 34.9% improvement over the strongest baseline, and, in aggregate, uniquely solves 46 tasks that no other agent could solve across all adopted advanced LLMs.

CCS Concepts

• **Software and its engineering** → **Automatic programming**; **Software testing and debugging**; • **Computing methodologies** → **Intelligent agents**.

Keywords

Large Language Models, Automated Program Repair, Autonomous Programming, Rust Benchmark

ACM Reference Format:
Jiahong Xiang[†], Wenxiao He, Xihua Wang, Hongliang Tian, and Yuqun Zhang[†*]. 2026. Evaluating and Improving Automated Repository-Level Rust Issue Resolution with LLM-based Agents. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773108

1 Introduction

The Rust programming language has become increasingly popular by offering a compelling combination of performance and safety [20, 64, 77]. Through its strict type system and ownership model, it provides compile-time guarantees against memory faults and data races [78], which has led to its wide adoption in critical domains such as operating systems [43, 57, 61], cloud services [9], and web browsers [32, 49, 80]. However, its core strengths, i.e., memory and thread safety, also cause its steep learning curve and coding difficulty, posing significant challenges for developers, with 83% of them finding it difficult to use [79] and 42% worrying about the long "time to productivity" [27]. Therefore, automating the resolution of the issues related to development in Rust is rather essential for unlocking Rust's full potential and extending its adoption.

Recently, Large Language Models (LLMs) have demonstrated powers in addressing a wide range of software engineering challenges [29, 33, 35, 45, 63, 66–69, 90]. Specifically, LLM code agents, which leverage LLMs' coding abilities through tool adoption, command execution, environmental feedback, and action planning, have demonstrated superior performance in code-related tasks [13, 87, 92] and aided developers in coding efficiency [15, 70]. Their

[*] Yuqun Zhang is the corresponding author.

[†] These authors are also affiliated with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773108

strengths are typically exemplified by their performance on popular benchmarks such as SWE-bench Verified [33], a version of SWE-bench crafted by OpenAI [54] for resolving real-world Python issues. In this benchmark, an agent, given only an issue description and a full repository codebase, is expected to autonomously generate a single corrective patch to resolve the issue and pass all tests in one attempt. This task is highly demanding, requiring a suite of capabilities including repository-level code understanding, targeted search, issue-reproducing test generation, and precise code editing. Surprisingly, the resolution rate on this benchmark has been improving dramatically, from a mere 1.2% achieved by approaches like RAG with SWE-Llama 13B in October 2023 [33] to a recent high 80.2% achieved by agents like Claude-Tools powered by the Claude 4 Opus model [13].

The demand for automating Rust issue resolution and the demonstrated power of advanced LLMs and code agents altogether raise a compelling question: *can agentic approaches effectively automate the resolution of real-world Rust issues?* To answer this question, it is essential for a large-scale, repository-level benchmark for real-world software engineering tasks for Rust and a comprehensive study on how advanced LLMs and agents perform on such tasks upon the benchmark. However, existing benchmarks for Rust are deficient in two key aspects. First, while existing Rust benchmarks focus on granular tasks such as C-to-Rust transpilation [14, 36, 56], function-level code synthesis [17, 50], or specific CVE analysis [52, 59, 91, 101], they are limited in involving general, repository-level software engineering problems. Second, existing large-scale SWE benchmarks predominantly feature Python and Java [33, 46, 98]. The few that incorporate Rust only offer a smaller corpus of tasks (e.g., 43 in SWE-bench Multilingual [37] and 239 in Multi-SWE-bench [96] compared with 500 Python issues in SWE-bench Verified [54]). Consequently, the absence of a benchmark that is both large-scale and dedicated to real-world Rust software engineering tasks prevents us from comprehensively assessing the effectiveness of the LLMs and agents on resolving practical, repository-level Rust issues.

To bridge this gap, we introduce Rust-SWE-bench, a large-scale benchmark that contains 500 real-world, issue-resolving repository-level Rust tasks from 34 popular repositories. Specifically, we adopt the well-established SWE-bench workflow [33] and adhere to the verification protocol established by the popular SWE-bench Verified [54] to construct it. Our data collection process begins by sourcing pull requests (PRs) from approximately 87 prominent open-source Rust repositories selected based on their popularity on GitHub (> 1,000 stars). We then identify candidate tasks with PRs that are both merged and linked to a resolved GitHub issue and that contain modifications to test files. Finally, to ensure that each task represents a verifiable fix, we validate whether each patch induces a "fail-to-pass" transition (i.e., at least one test fails before the patch and passes after the patch is applied), yielding a candidate set of 1,040 Rust software engineering tasks. To mitigate issues like underspecified problem descriptions [54, 89] which can lead to the underestimation of LLM capabilities and inefficient use of computational resources, we perform a final manual inspection of task candidates, following OpenAI's construction principles [54]. Eventually, we obtain 500 high-quality software engineering tasks for Rust from 34 diverse GitHub repositories.

To assess the capabilities of advanced LLMs and agents on this new benchmark, we conduct a comprehensive empirical study involving four representative agentic approaches (i.e., SWE-agent [92], OpenHands+CodeAct v2.1 [87], Agentless [89], and AutoCodeRover v2.0 [63]) and four state-of-the-art LLMs (i.e., Claude-Sonnet-3.7 [11], GPT-4o [53], OpenAI o4-mini [55], and Qwen3 [10]). Specifically, these studied agents are selected for their general-purpose design, i.e., having demonstrated capabilities across various programming languages [37, 60, 96]. The evaluation results demonstrate that the adoption of a ReAct-style [93] ('thought-act-observe' loop) architecture excels in solving tasks on Rust-SWE-bench. The associated agent OpenHands with Claude-Sonnet-3.7 [11] resolves a substantial 21.2% of real-world repository-level Rust issues, for which the manual resolution process averages 126 days and approximately 5.5 rounds of discussion. Moreover, we find that top-performing agents distinguish themselves by their ability to generate large, complex patches, e.g., the leading agent OpenHands resolves significantly more issues than Agentless (105 vs. 57) when the required patch size exceeds 150 lines. We also find the compilation errors stem from failures to model repository-wide code structure and to comply with Rust's strict type and trait semantics. Interestingly, we observe that issue reproduction is critical for Rust issue resolution, e.g., the remaining 44.5% of tasks fail at the reproduction stage and thus could not be successfully resolved even under the top-performing agent-model configuration.

To mitigate the issues of repository-wide code analysis and issue reproduction in our study findings, we further introduce RustForger, a novel agentic approach through automated test environment setup coupled with a cross-project dynamic tracing mechanism. Specifically, RustForger first constructs an isolated testing workspace, managing Cargo dependencies [24] to create a reliable and sandboxed environment for issue reproduction. Within this controlled environment, the agent can invoke a novel `Trace` command for cross-project dynamic analysis. This command automates the entire dynamic analysis workflow by leveraging Rust's metaprogramming capabilities—specifically procedural macros—to automatically instrument target functions through Abstract Syntax Tree (AST) modification. Such a decoupled strategy enables the agent to capture precise runtime control-flow information by executing tests from the clean workspace, thereby bypassing the complex build systems of the original project. Accordingly, RustForger equips the agent with the necessary runtime insights to more effectively diagnose and resolve complex real-world Rust issues. The evaluation results show RustForger solves 46 tasks that cannot be solved by any studied agent across all LLMs; its pairing with Claude-Sonnet-3.7 resolves 28.6% of tasks—a 34.9% improvement over the strongest baseline.

In summary, this paper makes the following contributions:

- **Benchmark.** We construct Rust-SWE-bench, the first large-scale, repository-level benchmark dedicated to real-world Rust software engineering issues. Comprising 500 high-quality, verified tasks from 34 popular repositories, Rust-SWE-bench addresses a critical gap in the field and enables systematic evaluation and future research on automated Rust development.

- **Study.** We conduct a comprehensive empirical study on Rust-SWE-bench with four representative agents and four state-of-the-art LLMs. Our analysis reveals that while ReAct-style agents show promising results, they are limited by two key challenges: comprehending repository-wide code structure and semantics, and successfully reproducing issues.
- **RustForger.** We propose RustForger, a novel agentic framework that addresses identified challenges by integrating an automated testing workspace with a cross-project dynamic tracing mechanism, powered by the distinctive programming features of Rust. Our evaluation shows that RustForger resolves 28.6% of tasks in the Rust-SWE-bench benchmark using Claude-Sonnet-3.7, significantly outperforming other baselines, and uniquely resolves 46 tasks across all adopted LLMs.

2 Background & Related Work

2.1 Rust Programming Language

Rust offers a modern solution for high-performance and secure software that provides strong compile-time guarantees against memory faults and data races without the overhead of a garbage collector. This combination of safety and performance has spurred its adoption in critical domains, including operating systems such as Redox [62] and Tock [88], web browsers such as Servo [83], and cloud infrastructure like Firecracker [9]. Notably, Rust now constitutes 21% of all new native code in the Android OS codebase [42].

While Rust's safety guarantees are enforced by its unique type system (built on ownership, borrowing, and lifetimes), such a system is notoriously difficult for newcomers to master, creating a steep learning curve and a significant barrier for adoption [22, 79]. To address this high development overhead, researchers and developers have proposed various approaches. RustAssistant [23] leverages LLMs to help developers automatically fix compilation errors. The LLM-driven Syzygy [65] aims to automate the migration of legacy C codebases to Rust. Concrat [30] analyzes and replaces unsafe C lock APIs in concurrent programs with provably safe Rust equivalents. Bronze [21] introduces an optional garbage collector to reduce the difficulty of complex memory management. PanicKiller [52] provides the first automated tool for fixing panic bugs in real-world Rust programs. However, despite these specialized solutions, a general-purpose agent capable of resolving diverse, real-world Rust issues remains an open challenge.

2.2 Large Language Model-based Agents

LLM-based agents refer to autonomous systems that use an LLM as a central controller to perceive and act upon an environment to achieve specific goals [34, 84]. These agents operate in an iterative cycle, leveraging core components: planning for decomposing complex tasks, memory for learning from historical observations [99], and perception for processing environmental feedback, as shown in Figure 1. Crucially, their action component enables them to utilize external tools, allowing them to interact with and modify their environment in ways far beyond simple text generation [84].

Researchers have attempted to use agents to automatically resolve real-world, repository-level software engineering tasks [26, 33, 37, 60, 95–97]. Specifically, SWE-bench [33] has gained significant attention since its release, with numerous agents being evaluated on this benchmark. In this benchmark, an agent, given only an issue description and the repository codebase, is expected to autonomously generate a single corrective patch to resolve the issue and pass all tests in one attempt. This task is highly demanding, requiring a suite of capabilities including repository-level code understanding, targeted search, issue-reproducing test generation, and precise code editing. Notably, reproducing an issue—whether a bug, a missing feature, or an API behavior—is a critical step for capturing dynamic execution information, which in turn facilitates more precise issue-related code localization and targeted patch generation [41] and serves as a foundational stage in recent advanced agents [19, 63, 87, 89, 92].

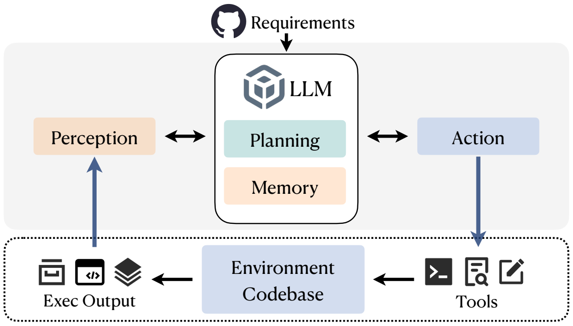


Figure 1: Basic Framework of LLM-based Agents

To resolve the tasks mentioned above, a variety of paradigms have been proposed. ReAct [93] enables an agent to synergize reasoning and acting by interleaving a 'thought-act-observe' loop. For instance, SWE-agent [92] implements the ReAct loop with an Agent-Computer Interface (ACI) by equipping LLMs with high-level tools for file searching, editing, and navigation, instead of raw terminal interaction. Similarly, the OpenHands agent [87], built upon the ReAct-style CodeAct architecture [85], operates as a generalist platform where agents perform tasks by executing code or conversing with humans for clarification. Another group of agents employ more structured, multi-stage workflows. For instance, AutoCodeRover v2.0 [63] utilizes a pipeline of specialized agents for iterative context retrieval and specification inference to guide patch generation. Agentless [89] demonstrates the efficacy of a simpler, non-iterative workflow (e.g., localize, repair, and validate) that deliberately limits the LLM's autonomy. While these agents show promising results on Python-centric benchmarks like SWE-bench Verified (e.g., OpenHands at 70.4% and Agentless at 50.8%), their effectiveness on real-world Rust issues remains underexplored.

2.3 SWE-related Benchmarks

SWE-bench [33] is a prominent benchmark for evaluating end-to-end software maintenance capabilities, comprising 2,294 real-world tasks derived from GitHub issues across 12 popular Python repositories. Specifically, the objective for each task is to automatically resolve the corresponding issue and submit a patch. The construction of SWE-bench follows a three-stage pipeline to ensure task quality: (1) scraping pull requests from well-maintained repositories, (2) filtering for those that resolve a specific issue and modify corresponding tests, and (3) retaining only instances that pass a "fail-to-pass" execution cycle, confirming the tests' relevance to the issue. In each task, an agent is provided with an issue description and the entire repository as the codebase, and is challenged to autonomously generate a corrective code patch. A submission is deemed successful only if the patch successfully applies and passes all unit and integration tests, which are hidden from the agent during the resolution task. To facilitate more reliable evaluation and reduce evaluation overhead [89], SWE-bench Verified provides a subset of 500 human-validated tasks with well-scoped tests and unambiguous issue descriptions [54].

Recently, researchers have extended the repository-level issue-resolution tasks for SWE-bench in a multilingual manner. For instance, Multi-SWE-bench [96] introduces a large-scale, human-annotated dataset of 1,632 tasks across seven languages, including Java, JavaScript, and Rust. SWE-bench Multilingual [37] offers a more compact set of 300 high-quality tasks across nine languages, designed for rapid evaluation while maintaining full compatibility with the original SWE-bench infrastructure. SWE-PolyBench [60] focuses on languages like Java and TypeScript. These benchmarks collectively represent a significant step towards evaluating the generalization capabilities of LLM-based agents across diverse software ecosystems. However, real-world Rust issues remain sparse in these benchmarks, featuring a small corpus of tasks (43 in SWE-bench Multilingual and 239 in Multi-SWE-bench vs. 500 in SWE-bench Verified). Hence, there is an urgent need to construct a large-scale Rust real-world SWE benchmark to facilitate a rigorous and systematic evaluation of agent capabilities within the Rust ecosystem.

3 Rust-SWE-bench

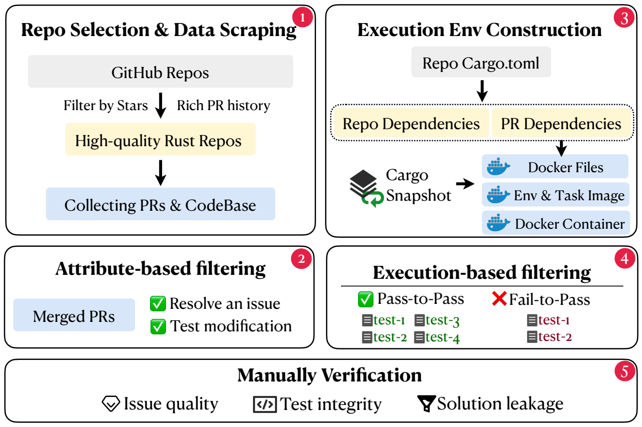


Figure 2: Rust-SWE-bench Construction Process

3.1 Benchmark Construction

Figure 2 shows a five-step construction process of Rust-SWE-bench, delineated as follows.

3.1.1 Repository selection and data scraping. We first select 87 popular open-source Rust repositories that meet several criteria: they must have over 1,000 GitHub stars, be well-maintained, feature an extensive test suite, and possess a rich history of pull request (PR) discussions. Specifically, we scrape approximately 80,000 merged PRs from these repositories, along with the codebase state at each PR's corresponding base commit.

3.1.2 Attribute-based filtering. We then apply an attribute-based filter to the scraped PRs, retaining only those that meet two criteria: being explicitly linked to a GitHub issue and introducing modifications to the project's test suite. Specifically, we conceptually divide the changes within each pull request's patch into two components: a test patch, which modifies the test suite, and a fix patch, which resolves the corresponding issue.

3.1.3 Execution environment construction. To ensure build reproducibility and mitigate dependency conflicts, we construct a snapshot-based execution environment for each task by configuring Cargo [24] to use a crates.io index snapshot at the time of the original pull request. Next, we analyze and classify dependencies as either repository-level or PR-specific to further minimize both storage overhead and the build time for each task's Docker image and container.

3.1.4 Execution-based filtering. Our validation process first identifies the `fail` state of tests, i.e., we apply only the PR's test patch to the base commit and execute the test suite. Next, we apply the PR's fix patch and re-execute the test suite to identify the `pass` state. A task is validated as a genuine fail-to-pass instance only if it meets three conditions: (1) at least one test exhibits a clear transition from fail to pass, (2) no previously passing tests (pass-to-pass tests) regress, and (3) the complete patch (both test and fix) introduces no new build failures, runtime errors, or test regressions. Specifically, unlike Python, to rigorously define a `fail` in Rust's compiled environment, we perceive compilation errors as test failures. This ensures that any non-compiling state, whether after applying the test patch or the fix patch, is correctly categorized as a failure. Accordingly, we obtain a pool of 1,040 candidate tasks.

3.1.5 Manual verification. We perform a rigorous manual verification of each candidate task instance, inspired by SWE-bench Verified [54] and AGENTLESS [89]. We assess each task's quality across three key dimensions: (1) issue quality, i.e., ensuring the problem description is sufficient and unambiguous; (2) fail-to-pass integrity, i.e., confirming the fail-to-pass behavior is deterministic and non-trivial; and (3) solution leakage, i.e., verifying that the issue does not contain explicit solutions. Our manual inspection details are provided on our GitHub page [3]. Notably, we label and classify each validated task by its primary type and challenge (as in Section 3.2). Eventually, our construction process yields the Rust-SWE-bench dataset, which comprises 500 high-quality tasks from 34 popular open-source Rust repositories, i.e., aligning with 500 tasks in the established SWE-bench Verified [54] for Python, with each task representing a well-defined and realistic software engineering problem.

3.2 Characteristics of Rust-SWE-bench

Rust-SWE-bench provides 500 high-quality, manually reviewed task instances from 34 repositories, making it considerably larger and more diverse than other existing Rust SWE benchmarks like SWE-bench Multilingual [37] (43 instances from 7 repositories) and Multi-SWE-bench [96] (239 instances from 10 repositories). To

Table 1: Characteristics of Repositories in Rust-SWE-bench

Repository	#GitHub Stars	Functionality	#Instance
`ripgrep`	53.6k	Command-line search tool	7
`ast-grep`	9.2k	CLI for code structural search	6
`async-graphql`	3.5k	GraphQL server-side library	14
`cargo-dist`	1.7k	Application packaging for Rust	10
`aya`	3.7k	eBPF library for Rust	6
`bevy`	40.5k	Data-driven game engine	46
`bincode`	3.1k	Binary encoder/decoder	8
`biome`	19.9k	Web project toolchain/linter	53
`boa`	5.7k	Embeddable JavaScript engine	49
`cargo-generate`	2.2k	Project template developer tool	12
`chrono`	3.6k	Date and time library	17
`clap`	15.3k	Command Line Argument Parser	56
`async-trait`	2k	Async functions in traits	9
`autocxx`	2.4k	Safe C++ from Rust interop	5
`bandwhich`	10.6k	CLI network utilization display	2
`cargo-edit`	3.2k	Manage cargo dependencies	11
`cbindgen`	2.7k	Generates C bindings from Rust	14
`nushell`	35.7k	A new type of shell	5
`rayon`	11.9k	Data parallelism library	2
`angle-grinder`	3.6k	Command-line log slicing tool	3
`askama`	3.6k	Jinja-like template engine	25
`cargo-fuzz`	1.7k	Command-line fuzzer for Rust	6
`cc-rs`	2.0k	Compile C/C++ in build scripts	8
`chalk`	1.9k	Rust trait system library	7
`cargo-make`	2.8k	Task runner and build tool	2
`serde`	9.8k	Serialization/deserialization	1
`bat`	53.3k	A 'cat' clone with highlighting	10
`fd`	38.6k	User-friendly alternative to 'find'	12
`cairo`	1.8k	Language for provable programs	9
`axum`	22.2k	Web application framework	19
`bytes`	2.1k	Utilities for working with bytes	22
`tokio`	29.0k	Asynchronous runtime for Rust	17
`tracing`	6k	Application-level tracing	13
`burn`	11.5k	Flexible Deep Learning Framework	14

Table 2: Overall Statistics of the Rust-SWE-bench

Component	Metric	Mean	Median	Max
Issue Text	Length (Tokens)	393.4	245.5	3,826
Codebase	# Files	993.6	335	15,036
	# Lines	128,126	66,069	753,715
Fix Patch	# Files edited	9.8	4	148
	# Hunks edited	9.9	4.5	148
	# Lines edited	139.9	40.5	15,449
Tests	# Fail to Pass	96.5	5	1,733
	# Pass to Pass	219.6	14	1,780

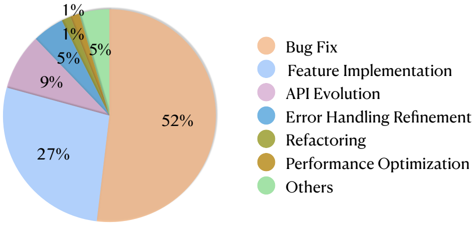


Figure 3: Distribution of Task Categories in Rust-SWE-bench

ensure broad functional coverage, Rust-SWE-bench incorporates diverse project domains as shown in Table 1, including command-line utilities (e.g., `ripgrep` [28], `fd` [58]), development tools (e.g., `cargo-edit` [38], `biome` [16]), concurrency and parallelism libraries (e.g., `tokio` [82], `rayon` [73]), web frameworks (e.g., `axum` [81]), and specialized engines (e.g., `bevy` [71], `boa` [72]).

Table 2 presents the overall statistics of Rust-SWE-bench, highlighting its scale and task complexity. Specifically, the Rust-SWE-bench demonstrates significant diversity in codebase size. On average, its projects contain 993.6 files and 128,126 lines of code, with the largest repository reaching over 15,000 files and 750,000 lines. This scale ensures that Rust-SWE-bench effectively represents a wide spectrum of real-world software development scenarios. Moreover, a key characteristic of Rust-SWE-bench is the high complexity of its tasks, requiring an average of 9.8 files, 9.9 hunks, and 139.9 lines of code to resolve an issue, whereas a fix in Python-based SWE-bench Verified [54] requires on average 1.25 files, 2.46 hunks, and 14.32 lines. This contrast implies Rust-SWE-bench potentially better reflects real-world software engineering complexities.

To characterize the nature of the tasks, we manually classified all 500 instances. Figure 3 shows their distribution where 52% of the tasks are Bug Fix issues, followed by 27% Feature Implementation issues, indicating that Rust-SWE-bench mainly focuses on practical software maintenance and evolution workflows. The high code churn detailed in Table 2 (averaging 139.9 lines edited vs. 14.3 in SWE-bench Verified) suggests that complex code modifications are a common feature of routine maintenance in Rust. The benchmark also includes categories that are highly relevant to Rust's ecosystem, e.g., API Evolution (9%) and Error Handling Refinement (5%), reflecting the language's evolving APIs and emphasis on explicit error management [44]. Note that their original manual resolution takes on average 113 days from issue creation to PR merge and involves approximately 7.2 rounds of developer discussion. We further analyze the distribution of these resolution durations and find that 14.0% of tasks are resolved within a day, 26.6% within a week, 21.6% within a month, and 21.8% up to six months, while 16.0% exceed six months. This indicates that Rust-SWE-bench covers real-world scenarios, spanning from quick fixes to long-standing issues.

4 The Extensive Study

4.1 Study Setup

4.1.1 Study Subjects. We evaluate four representative SWE agents including SWE-agent [92], OpenHands+CodeAct v2.1 [87], Agentless [89], and AutoCodeRover v2.0 [63] on our Rust-SWE-bench benchmark with their details as follows.

- SWE-agent: SWE-agent designs an agent-computer interface which defines the possible actions taken by an agent to edit code, navigate the codebase, and execute tests.
- OpenHands+CodeAct v2.1: OpenHands utilizes the CodeAct [85] architecture, enabling it to perform tasks by executing code-based actions.
- Agentless: Agentless introduces a structured, multi-stage workflow (e.g., localize, repair, and validate) that deliberately constrains the LLM's autonomy to a predefined sequence.
- AutoCodeRover v2.0: AutoCodeRover v2.0 employs an iterative process of context retrieval and specification inference to progressively guide the generation of code patches.

4.1.2 LLMs. We adopt four advanced LLMs including Claude-Sonnet-3.7 (claude-3-7-sonnet-20250219) [11], GPT-4o (gpt-4o-2024-11-20) [53], OpenAI o4-mini (o4-mini-2025-04-16) [55], and Qwen3 (qwen3-235b-a22b) [10]. We obtain the open-source model Qwen3 from Hugging Face [1] and access Claude-Sonnet-3.7, GPT-4o, OpenAI o4-mini through the APIs provided by Anthropic [12] and OpenAI [2]. Inference for the open-source model Qwen3 is conducted on servers with 128-core 2.6GHz AMD EPYC™ ROME 7H12 CPU, 512 GiB RAM, and eight NVIDIA A100 80GB GPUs, running Ubuntu 20.04.6 LTS, utilizing vllm [40] inference framework.

4.1.3 Evaluation Metrics. Following prior works [31, 33, 48, 96], our major evaluation metric is the **Resolved Rate (%)**, reported as **Pass@1** efficacy. An issue is considered resolved if a single generated patch successfully applies to the codebase and passes all developer-written acceptance tests. Crucially, these acceptance tests are held out and not used by the agent during the patch generation process to ensure a fair evaluation. Moreover, we report the average API inference cost (**$ Avg. Cost**) and token usage (**# Avg. Token**). Note that to calculate the cost of the open-source LLM Qwen3, we adopt the pricing model from its official API webpage [10]. Furthermore, to specifically evaluate the issue reproduction stage adopted by all our four studied agents in RQ2, we introduce an additional metric: **Reproduction Success Rate (%)**, which measures the ratio of tasks where the agent-generated test successfully replicates the behavior described in the original issue.

4.2 Implementations

4.2.1 Agent Adaptation for Rust. Following the prior work [96], we adapt the prompts of our four studied agents for the Rust language. Moreover, agents like AutoCodeRover and Agentless include an issue reproduction stage. However, reproducing issues in Rust requires using shell commands to manage the environment and dependencies of Cargo projects. Therefore, we develop custom shell tools to enable this stage for these agents on Rust-SWE-bench. Furthermore, we adapt AutoCodeRover's program structure-aware APIs from Python to Rust, enabling the agent to gather relevant code context. For other configurations, including decoding parameters (e.g., temperature and top-p), we adhere to the original setups. The detailed implementations are shown in our GitHub page [3].

4.2.2 Evaluation of the Agent's Reproduction Stage. To assess the agents' capabilities in reproducing issues, we extract the necessary commands and files from their issue-resolving trajectories and re-execute them within a sandboxed Docker container to capture the execution results. Specifically, we first run the reproduction test to verify its syntactic validity [51, 86]. However, due to the complexity of Rust issues [18], we cannot analyze the reproduction results directly via assertions as in prior work [51, 86]. Therefore, for tests meeting this criterion, we then assess the reproduction results via manual analysis of the test execution output (including error messages and exit codes) to determine if the observed behavior corresponds to the problem described in the original issue.

4.3 Research Questions

We investigate the following research questions to study the effectiveness of the agentic approaches and the factors that impact their effectiveness on Rust-SWE-bench.

- **RQ1:** *How do different agents perform on Rust-SWE-bench?* For this RQ, we benchmark the selected agents-model configurations on Rust-SWE-bench and report their overall effectiveness.
- **RQ2:** *What are the behavioral characteristics of agents resolving real-world Rust issues?* For this RQ, we perform a detailed behavioral analysis of the agents, which involves examining the scope of code edits, the types of compilation errors they produce, and the tests they reproduce.

4.4 Result Analysis

4.4.1 RQ1: performance of agents and models on Rust-SWE-bench. Table 3 presents the resolution rates for each agent-model configuration on Rust-SWE-bench. We find that agents built upon the ReAct paradigm [93], such as OpenHands and SWE-agent, achieve the best performance on this benchmark. Specifically, the top-performing agent-model configuration, OpenHands paired with Claude-Sonnet-3.7, resolves 106 (21.2%) tasks. When resolved manually, these tasks typically require an average of 126 days from issue opening to PR merge and 5.5 rounds of discussion. Notably, this result largely outperforms not only the runner-up ReAct-based agent SWE-agent (15.0% tasks), but also the runner-up LLM with its own framework OpenAI o4-mini (6.8%). These results underscore the decisive and compounding impact of both the ReAct-style agentic architecture and the choice of the underlying language model.

> *Finding 1: The adoption of a ReAct-style architecture achieves the best performance on Rust-SWE-bench.*

Moreover, we observe a significant cost disparity among the agent-model configurations. The top-performing agents, OpenHands and SWE-agent, are also the most expensive, e.g., costing $3.81 and $3.48 on average when paired with Claude-Sonnet-3.7, respectively. This higher cost is potentially caused by their ReAct-based [93], iterative problem-solving frameworks, which lead to a greater number of interaction rounds and consequently higher token consumption compared to the more structured, single-pass workflows of Agentless and AutoCodeRover.

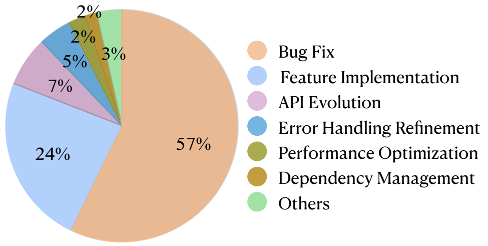


Figure 4: Distribution of All Resolved Tasks by Category.

Figure 4 shows the distribution of the resolved task types by all studied agents which closely aligns with the overall benchmark's composition in Figure 3, e.g., bug fixes (57%) and feature implementations (24%) dominate the resolved tasks. This alignment demonstrates that the agents are effective in tackling common challenges in the Rust ecosystem.

Table 3: Evaluation Results of Different Models and Agents in Rust-SWE-bench

Agent	LLM	% Resolved	#Edited			Avg.	Avg.
			Line	Hunk	File	$ Cost	# Token
OpenHands	Claude-Sonnet-3.7	106(21.20%)	146.03	5.59	3.84	3.81	1,236,942
	GPT-4o	33 (6.60%)	66.59	3.19	2.50	1.85	729,393
	OpenAI o4-mini	34 (6.80%)	20.25	2.32	1.42	1.23	1,106,811
	Qwen3	25 (5.00%)	81.05	3.00	2.30	0.35	492,141
SWE-agent	Claude-Sonnet-3.7	75(15.00%)	190.49	6.71	4.02	3.48	1,131,753
	GPT-4o	9 (1.80%)	172.84	5.28	3.45	1.92	755,343
	OpenAI o4-mini	41 (8.20%)	77.13	3.81	2.32	2.13	1,708,013
	Qwen3	9 (1.80%)	83.47	2.33	1.72	0.33	417,091
Agentless	Claude-Sonnet-3.7	36 (7.20%)	13.03	1.37	1.06	1.47	439,850
	GPT-4o	27 (5.40%)	16.45	1.93	1.24	0.89	331,373
	OpenAI o4-mini	39 (7.80%)	13.10	1.43	1.06	0.61	441,949
	Qwen3	16 (3.20%)	13.41	1.30	1.07	0.35	460,175
AutoCodeRover	Claude-Sonnet-3.7	46 (9.20%)	22.01	1.45	1.13	1.18	352,467
	GPT-4o	21 (4.20%)	15.01	1.53	1.24	1.09	369,833
	OpenAI o4-mini	34 (6.80%)	17.38	1.51	1.09	0.46	343,247
	Qwen3	24 (4.80%)	12.09	1.37	1.14	0.32	359,351

4.4.2 RQ2: behavioral analysis of agents on Rust-SWE-bench. Table 3 presents the edit scopes of patches generated by different agents. We find that agents like SWE-agent and OpenHands cause substantially larger modifications, editing on average over 2 files and 70 lines of code per task, while Agentless and AutoCodeRover typically edit 1.2 files on average and fewer than 20 lines.

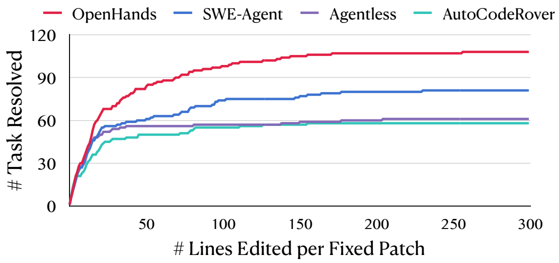


Figure 5: Cumulative Resolved Tasks by Edited Lines per Fixed Patch, Aggregated per Agent Across All LLMs.

We further analyze the capabilities of different agents on tasks requiring varying edit scopes, with results aggregated across all models for each agent, as shown in Figure 5. The result reveals that all four agents perform similarly on tasks solvable with small patches, e.g., OpenHands resolves 48 tasks and Agentless resolves a comparable 43 within a 15-line edit scope. In contrast, a clear performance gap appears as the required patch size increases. Within the 150-line scope, OpenHands resolves 105 tasks, while Agentless resolves only 57. Such results indicate that powerful agents like OpenHands and SWE-agent tend to explore a wider solution space by making more extensive changes.

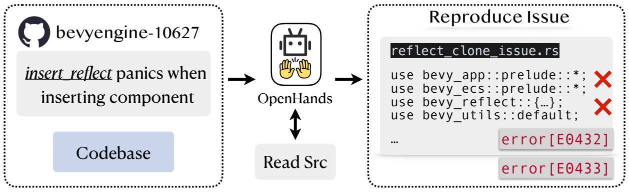
> *Finding 2: While all agents are similarly effective on tasks requiring small patches, top-performing agents are more powerful to generate large, complex patches for more challenging tasks.*

Table 4: Statistics of Common Rust Compilation Errors Encountered During Agent Issue Resolution

Error Code	Error Description	Rate (%)
E0599	A method is used on a type which doesn't implement it	18.06%
E0433	An undeclared crate, module, or type was used	16.21%
E0432	An import was unresolved	12.08%
E0425	An unresolved name was used	8.54%
E0308	Expected type did not match the received type	6.81%
E0277	Type does not implement expected trait	6.50%
E0412	A used type name is not in scope	5.69%
E0753	An inner doc comment was used in an invalid context	3.07%
E0282	The compiler could not infer a type and asked for a type annotation	1.75%
E0609	Attempted to access a nonexistent field in a struct	1.48%
E0061	An invalid number of arguments was passed when calling a function	1.23%
E0405	The code refers to a trait that is not in scope	1.21%
E0407	A definition of a method not in the implemented trait was given in a trait implementation	1.18%

We find agents frequently encounter compilation errors when resolving real-world Rust issues, e.g., OpenHands encounters an average of three compilation errors per task. We then examine the overall distribution of common compilation errors produced during their resolution process, detailed in Table 4. We find that the compilation errors originate from two major challenges. One is the failure to comprehend repository-wide code organization (43.7%), leading to errors in naming, scoping, and path resolution (i.e., `E0433`, `E0432`, `E0425`, `E0412`, and `E0405`). This suggests that agents struggle to correctly model the project's structural context for linking different code modules, which is a prominent challenge during the issue reproduction stage. For instance, in the case of `bevyengine-10627` [4] shown in Figure 6, OpenHands with Claude-Sonnet-3.7 attempts to construct a reproduction test to replicate a panic [39] where cloning a reflected trait object (`Box<dyn`

`Reflect>`) via the `clone_value()` method erases its type information, causing a subsequent call to `insert_reflect` to fail. However, compilation errors arise from unresolved imports and undeclared types, causing a failure to incorporate the necessary `bevy crate preludes` [71] and dependencies into the test's scope.



```
reflect_clone_issue.rs

use bevy_app::prelude::*;
use bevy_ecs::prelude::*;
use bevy_reflect::{…};
use bevy_utils::default;

…
error[E0432]
error[E0433]
```

Figure 6: Compilation Error in `bevyengine-10627` Task

Another major challenge for compilation errors (32.6%) arises from Rust's complex type and trait system. While agents often generate syntactically plausible code, they fail to satisfy the strict semantic contracts of Rust (i.e., `E0599`, `E0308`, `E0277`, and `E0407`). For example, in the case of `askama-374` [5], SWE-Agent's syntactically valid attempt to call a method on an object consistently triggered an `E0599` compilation error because the object's type failed to satisfy the necessary trait bounds, indicating the agent's difficulty to comply with Rust's strict semantics.

> *Finding 3: Agent-generated compilation errors primarily stem from two key deficiencies: a failure to model repository-wide code structure and an inability to comply with Rust's strict type and trait semantics.*

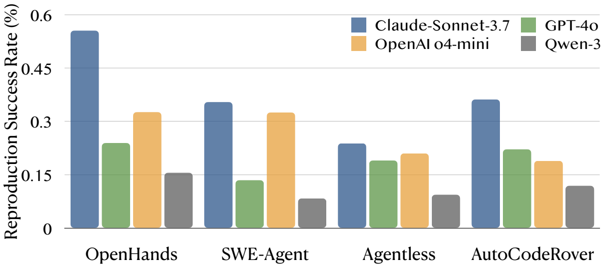


Figure 7: Reproduction Success Rate by Agents and Models

Motivated by Finding 3, we find that these compilation errors largely occur in the issue reproduction stage, e.g., 35% of them occur during the reproduction stage for the top-performing OpenHands-Claude-Sonnet-3.7 configuration, indicating that it is challenging for agents to reproduce issues. Figure 7 presents our evaluation of the issue reproduction stage, which involves re-executing tasks from Rust-SWE-bench and analyzing their outputs. OpenHands paired with Claude-Sonnet-3.7 achieves the highest reproduction success rate at 55.5%, i.e., the remaining 44.5% of tasks fail at the reproduction stage, i.e., a critical prerequisite for validating patches and obtaining runtime information, and thus could not be successfully resolved even under the top-performing agent-model configuration. Specifically, we find that for each successfully reproduced issue, OpenHands with Claude-Sonnet-3.7 re-runs the reproduction test on average 4.7 times for patch validation and dynamic information gathering. Moreover, we also disable the reproduction stage in OpenHands paired with Claude-Sonnet-3.7 and find that this causes a 42% drop in its resolution rate, indicating that issue reproduction is a prominent bottleneck for Rust issue resolution. To better understand these reproduction failures, we manually inspect the failed instances and find that most cases stem from dependency issues (52.5%; misconfigured `Cargo.toml` or workspace paths), followed by irrelevant tests (35.5%; tests that do not exercise the reported behavior) and non-compiling tests (12.0%; generated tests that fail to build).

> *Finding 4: Issue reproduction is critical for Rust issue resolution.*

5 Approach

Inspired by Finding 4, we introduce RustForger, a novel agentic approach to enhance Rust issue resolution through automated test environment setup coupled with a unique dynamic tracing strategy.

5.1 RustForger Framework

Figure 8 illustrates the framework of RustForger comprising two stages. In the first stage *Setup Testing Workspace*, the agent's primary objective is to create a reliable, isolated environment for issue reproduction. It begins by parsing the input issue description and performing an initial exploration of the target codebase to understand its module structure and dependencies, adhering to the established practices of ReAct-style agents [87, 92]. Subsequently, the agent automatically executes three key steps: (1) it analyzes the project's `Cargo.toml` to identify its structure and dependencies; (2) it initializes a new, clean Cargo project in a separate workspace; and (3) it imports the original, target project as a local path dependency into this new workspace. This ensures that any test will run against the exact code under investigation. Finally, within this isolated workspace, the agent generates a test case to specifically trigger and reproduce the failure described in the issue.

Once the issue is successfully reproduced, the agent launches the second stage, *Code Analysis & Patch Generation*, where it leverages the controlled environment for dynamic analysis. After using the test outputs to locate potential issue-related code regions, it activates its core cross-project dynamic tracing capability via the `Trace` command, a unified agent-computer interface that encapsulates the entire analysis workflow into a single, all-in-one function. This process is highly automated, powered by Rust metaprogramming features [74–76]: RustForger first injects the necessary tracing dependencies into the test environment. It then instruments targeted functions within the original codebase by injecting tracing macros via Abstract Syntax Tree (AST) modification. Critically, the test execution and data collection are launched from the testing workspace, as indicated by the dotted arrow in Figure 8. This decoupled strategy allows the agent to capture precise, runtime control-flow information without being entangled in the original project's complex build system or test suite. The collected trace data provides the potential path to the issue's origin, enabling the agent to propose a code modification. This patch is then iteratively verified against the reproduction test until the issue is resolved and

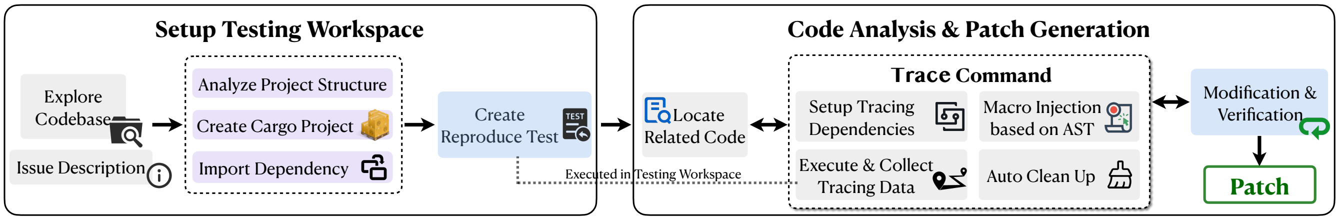

Setup Testing Workspace
Explore Codebase
Issue Description
Analyze Project Structure
Create Cargo Project
Import Dependency
Create Reproduce Test
TEST
Executed in Testing Workspace
Code Analysis & Patch Generation
Locate Related Code
Trace Command
Setup Tracing Dependencies
Macro Injection based on AST
Execute & Collect Tracing Data
Auto Clean Up
Modification & Verification
Patch

Figure 8: The RustForger Framework

the agent outputs the final patch. Notably, RustForger's design is hybrid: it prioritizes the dynamic analysis powered by the Trace command for reproducible issues, while utilizing the static analysis for all other cases, i.e., the agent bypasses dynamic analysis and proceeds directly to the *Code Analysis & Patch Generation* stage, relying solely on static analysis to locate and modify the code without using the Trace command.

5.2 Trace Command

In this section, we introduce the Trace command for root cause analysis in complex Rust projects. It provides a programmatic interface to a cross-project tracing workflow, which automates code instrumentation, test execution, and runtime data collection. Specifically, the complete workflow of the Trace command integrates three key components: a selective AST-based instrumentor, a robust tracing runtime with intelligent data serialization, and a unified agent-computer interface for seamless agent integration.

Selective AST-based Instrumentation. To capture execution flow and ensure syntactic integrity, RustForger's Trace command employs a selective instrumentor that directly modifies the source code's Abstract Syntax Tree (AST). It injects tracing macros at specified target functions. Specifically, the process is decoupled from the build system, enabling instrumentation of the original project code while running tests in a separate, isolated workspace. To ensure robustness and manage complexity, our current implementation of the Trace command focuses on instrumenting regular functions, while automatically excluding more complex structures like closures, other procedural macros, and test functions to prevent potential tracing conflicts and irrelevant overhead.

Robust Tracing Runtime. Our tracing runtime reconstructs the dynamic call graph by capturing function input and output. However, this task is challenging due to Rust's type system, in which the non-serializability [25] of many types would otherwise cause compilation failures. To handle this, we adopt a hybrid tracing strategy that captures serializable types as JSON while recording descriptive placeholders for others, thereby preserving crucial type information without compromising build integrity. The resulting trace, encompassing the call hierarchy and captured data, is structured into a single JSON object to facilitate programmatic analysis.

Unified Agent-Computer Interface. The Trace command encapsulates our entire tracing workflow into a unified agent-computer interface, inspired by SWE-agent [92]. The agent invokes it with parameters defining the target instrumentation functions, execution command, the testing workspace path, and the target codebase path.

Table 5: Evaluation Result of RustForger

Agent	Claude-Sonnet-3.7	GPT-4o	OpenAI o4-mini	Qwen3
OpenHands	106(21.2%)	33(6.60%)	34(6.80%)	25(5.00%)
SWE-agent	75(15.0%)	9(1.80%)	41(8.20%)	9(1.80%)
Agentless	36(7.2%)	27(5.40%)	39(7.80%)	16(3.20%)
AutoCodeRover	46(9.20%)	21(4.20%)	34(6.80%)	24(4.80%)
RustForger	**143(28.6%)**	**42(8.4%)**	**82(16.4%)**	**33(6.60%)**

To guarantee a non-destructive operation, the Trace command automatically backs up the project state before instrumentation and restores it upon completion, usually within a second. This interface abstracts the underlying complexity, i.e., automating dependency setup, AST-based macro injection, test execution, data collection, and final cleanup (Figure 8), enabling the agent to focus on high-level resolution strategies instead of implementation details.

5.3 Evaluation

5.3.1 Evaluation Metrics and Baselines. To evaluate the performance of RustForger, we adopt metrics used in our extensive study (Section 4.1.3): Resolve Rate (%), $ Avg. Cost, and Reproduction Success Rate (%). Due to budget constraints, we follow the SWE-agent setup [92] to set the budget per task to $4. We adopt the same four studied agents and LLMs in our previous study as our baselines.

5.3.2 Implementation. RustForger is implemented as a ReAct-style agent [93] and follows the setup of baseline agents [63, 87, 92], with its temperature set to 0 to ensure the model's output is more deterministic. Notably, our Trace's instrumentation mechanism is designed as a purely observational wrapper to strictly preserve original program semantics. To ensure identical trait resolution, the injected macros retain original function signatures, including all generics and where clauses. Furthermore, we adhere to Rust's borrow checker by exclusively utilizing temporary, strictly-scoped immutable references to capture parameter values. This ensures that variable lifetimes and drop timings remain unaffected, with all macro-generated variables isolated within private internal scopes. Specifically, Trace's overhead is minimal: tokens generated for tracing account for only 1.65% of total usage, the shared isolated workspace's initial build takes about 40 seconds, and each Trace invocation adds only ~12 seconds of incremental compilation. Due to page limits, implementation details like RustForger's prompt, basic functions (e.g., string replacer) and Trace are shown in our GitHub page [3].

5.3.3 Result Analysis. Table 5 presents the evaluation results where RustForger, when paired with Claude-Sonnet-3.7, achieves the best performance on Rust-SWE-bench by successfully resolving 143 out of 500 tasks (28.6%), outperforming the best-performing agent in our study OpenHands with Claude-Sonnet-3.7 by 34.9%. Notably, the `Trace` command is successfully leveraged in the resolution process for 71.3% of these tasks. Moreover, we validate the statistical significance of this result using McNemar's test [47], a widely adopted statistical method for comparing paired binary outcomes in software engineering research. The test yields a p-value < 0.001 when comparing RustForger with the runner-up OpenHands, indicating that the performance gain is statistically significant. RustForger also consistently outperforms each baseline agent across all four tested LLMs. This consistent superiority underscores the robustness and general effectiveness of our proposed framework in tackling real-world Rust issues. Notably, RustForger with the cost-effective OpenAI o4-mini achieves a 16.4% resolution rate (82 tasks), surpassing not only all baselines using the same model but also the highly-regarded SWE-agent paired with the much more powerful Claude-Sonnet-3.7 (15.0%). Such a result indicates that by effectively addressing the core challenges of issue reproduction and runtime analysis, our approach enables even less powerful models to achieve highly competitive results on complex real-world repository-level Rust tasks. At last, across all LLMs, RustForger uniquely resolves 46 tasks that no baselines could.

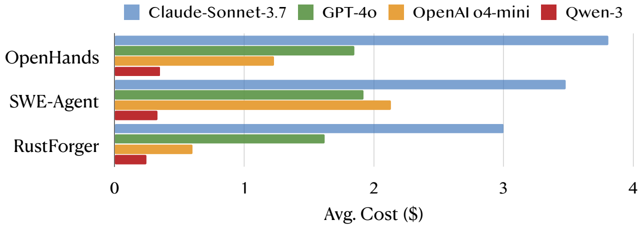


Figure 9: Comparison of Average Cost per Task

In addition to its superior resolution rates, RustForger also demonstrates significant advantages in cost efficiency. As shown in Figure 9, RustForger consistently incurs lower average API costs than the top-performing baselines, OpenHands and SWE-agent, across all LLMs. For instance, when paired with the most powerful model Claude-Sonnet-3.7, RustForger's average cost is $3.0, which is 21.3% and 13.8% lower compared to OpenHands ($3.81) and SWE-agent ($3.48), respectively. This gain is even more significant with more economical models–with OpenAI o4-mini, RustForger's cost ($0.6) is less than half that of OpenHands ($1.23) and amounts to only 28.2% of the cost of SWE-agent ($2.13).

Moreover, we examine how task complexity, approximated by the original manual resolution duration, correlates with agent performance. Our analysis shows that RustForger's success rate is 42.9% on issues resolved within a day, 32.3% within a week, 31.5% within a month, 22.0% within six months, and 15.0% for issues open longer than six months, highlighting both RustForger's strong performance and the particular difficulty of long-standing issues.

Figure 10 shows that RustForger consistently outperforms all baselines in terms of reproduction success rate across all LLMs, achieving a 67.3% rate with Claude-Sonnet-3.7 (vs. 55.5% for OpenHands) and a 45.4% rate with OpenAI o4-mini (vs. 32.6% for OpenHands).

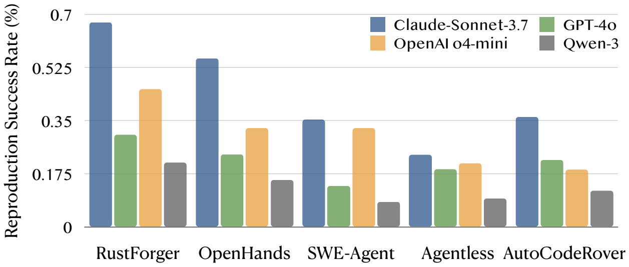
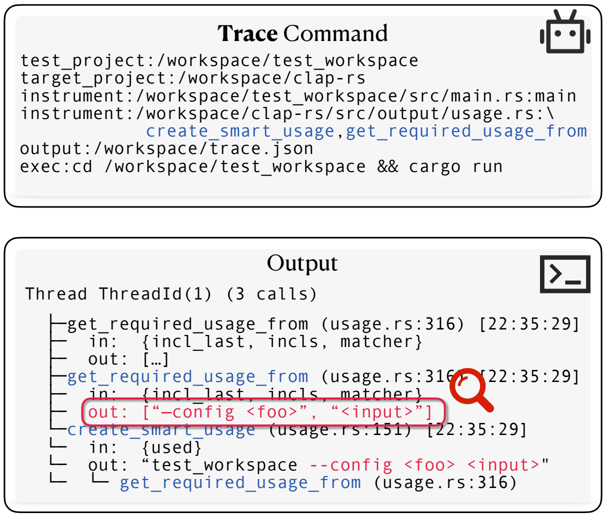


Figure 10: Comparison of Reproduction Success Rate


Trace Command

```
test_project:/workspace/test_workspace
target_project:/workspace/clap-rs
instrument:/workspace/test_workspace/src/main.rs:main
instrument:/workspace/clap-rs/src/output/usage.rs:\
                 create_smart_usage,get_required_usage_from
output:/workspace/trace.json
exec:cd /workspace/test_workspace && cargo run
```

Output

```
Thread ThreadId(1) (3 calls)

   ├─get_required_usage_from (usage.rs:316) [22:35:29]
   ├─  in:  {incl_last, incls, matcher}
   ├─  out: […]
   ├─get_required_usage_from (usage.rs:316) [22:35:29]
   ├─  in:  {incl_last, incls, matcher}
   ├─  out: ["–config <foo>", "<input>"]
   └─create_smart_usage (usage.rs:151) [22:35:29]
   └─  in:  {used}
   └─  out: "test_workspace --config <foo> <input>"
   └─  └─ get_required_usage_from (usage.rs:316)
```

Figure 11: The `Trace` Command Used in `clap-rs-2609` [6]

Case Study. To illustrate RustForger's practical advantages, we analyze its unique resolution of issue `clap-rs-2609` [6], a task where all baseline agents fail. The issue involves the popular `clap` library generating an incorrect usage string for arguments with default values. RustForger first executes its automated setup strategy, creating an isolated testing workspace and linking the original project as a local dependency, while other agents consistently fail to reproduce the issue due to the complex build configurations. Subsequently, the agent invokes the `Trace` command to perform cross-project dynamic analysis on the functions responsible for usage generation as in Figure 11. Specifically, `Trace` automates the entire workflow by: (1) setting up dependencies for both the `test_workspace` and the target codebase `clap-rs`; (2) instrumenting key functions, i.e., `main` in the test workspace, and `get_required_usage_from` and `create_smart_usage` in the `clap-rs` source code, with a tracing macro; (3) executing the test via `cargo run` and collecting the resulting trace information; and finally, (4) cleaning up all modifications. Interestingly, the trace data precisely expose the root cause: a faulty interaction where

Table 6: Ablation Study Results

Configuration	Claude-Sonnet-3.7	GPT-4o	OpenAI o4-mini	Qwen3	Avg.
RustForger$_{base}$	42	12	26	8	22.0
RustForger$_{reproduce}$	47	14	29	10	25.0
RustForger	**59**	**16**	**35**	**11**	**30.3**

one function get_required_usage_from passes an incorrect list of required arguments to create_smart_usage. This deep runtime insight enables the agent to correctly deduce a two-part patch modifying both functions. Finally, the agent iteratively validates its solution within the isolated workspace, leveraging the rapid feedback loop to efficiently converge on the correct solution.

5.4 Ablation Study

We further perform an ablation study on 200 randomly selected Rust-SWE-bench tasks, comparing the full RustForger against two variants: RustForger$_{reproduce}$, which disables the Trace command, and RustForger$_{base}$, which skips the isolated testing workspace entirely and directly performs the reproduction in the target codebase.

Table 6 presents the results of our ablation study. First, comparing RustForger$_{base}$ with RustForger$_{reproduce}$, the introduction of the isolated testing workspace alone improves the task resolution performance by resolving 3.0 more tasks on average. Furthermore, the performance leap from RustForger$_{reproduce}$ to the full RustForger underscores the value of the cross-project dynamic tracing analysis where the Trace command enables the agent to resolve an additional 5.3 tasks on average. Such results reflect the effectiveness of the issue reproduction and the dynamic tracing analysis of RustForger respectively.

To better understand where Trace helps, we also examine two representative issues. In bevyengine/bevy-16747 [7], RustForger must extend the animated_field! macro to support tuple structs (e.g., TextColor::0) while preserving the behavior for named fields. By invoking Trace on the underlying implementation methods, the agent verifies at runtime that calls for named fields still go through new_unchecked, whereas tuple indices are correctly dispatched to new_tuple_unchecked, allowing it to validate the updated macro's semantics beyond static inspection. In tokio-rs/tracing-1017 [8], RustForger uses Trace to validate a performance optimization that skips Thread-Local Storage checks when no scoped dispatcher is active. By tracing set_global_default and with_default in the reproduction test, it observes that the global dispatcher is initialized before scoped guards, providing runtime evidence that the conditional bypass does not compromise correctness or thread-safety. These cases concretely illustrate how dynamic tracing contributes to the gains observed in the ablation study.

6 Threats to Validity

Threats to internal validity. We take several measures to ensure the internal validity of our study. To guarantee reproducibility and mitigate system-level variations, we conduct each experimental run in a dedicated Docker container that encapsulates the exact repository state, toolchain, and dependencies. Furthermore, our Trace command implements a meticulous cleanup mechanism to remove all instrumentation artifacts after its execution, regardless of the outcome, ensuring our analysis introduces no residual side effects to the target codebase.

To mitigate the inherent stochasticity of LLM-based agents, we perform our evaluation across 500 diverse instances in Rust-SWE-bench, ensuring the stability and generalizability of our findings. We set the temperature to 0 to mitigate the threat of randomness to the RustForger evaluation results. Potential bias in our manual validation is mitigated through a strict protocol where three authors independently verify each result before reaching a consensus.

Threats to external validity. The primary threat to external validity lies in the generalizability of our findings, which is closely tied to our evaluation dataset and the selection of compared agents. Correspondingly, we construct Rust-SWE-bench with 500 real-world, issue-resolving tasks sourced from 34 diverse and popular open-source Rust repositories. Each task is grounded in an actual merged pull request, ensuring the problems are authentic and representative of genuine software engineering challenges. Meanwhile, we select four recent and representative agents that demonstrate state-of-the-art performance on the widely-recognized SWE-bench benchmark, ensuring our analysis reflects the latest landscape of code agents.

Threats to construct validity. A potential threat to construct validity lies in whether our evaluation metrics accurately capture the multifaceted nature of the issue resolution task. To mitigate this, we adopt the Resolved Rate (Pass@1), a widely-accepted metric for evaluating task completion in code generation benchmarks [31, 33, 48, 96]. To provide deeper diagnostic insights, we also measure the Reproduction Success Rate (%) [94, 100] to assess an agent's ability to replicate the original issues.

7 Conclusion

In this paper, we introduce Rust-SWE-bench, the large-scale, repository-level benchmark for real-world Rust software engineering issues, comprising 500 issue-resolving tasks from diverse and popular repositories. Our comprehensive study on Rust-SWE-bench reveals that the performance of current LLM-based agents is primarily hindered by challenges in repository-wide code comprehension and issue reproduction. To address these limitations, we design RustForger, a novel agentic framework integrating an automated testing workspace with a unique cross-project dynamic tracing capability. Our evaluation demonstrates that RustForger achieves the best performance, resolving 28.6% of the tasks using Claude-Sonnet-3.7, i.e., a 34.9% improvement over the strongest baseline, and uniquely resolving 46 issues across all adopted LLMs.

Data Availability

All study results, evaluation details, and source code of the Rust-SWE-bench and RustForger are presented in the *GitHub* page [3].

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372220). It is also partially supported by Ant Group Research Fund.

References

[1] 2024-02-29. Hugging Face. https://huggingface.co.

[2] 2024-02-29. OpenAI API. https://openai.com/api.

[3] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench.

[4] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench/blob/main/material/MopenHands_claude_evyengine__bevy-10627.traj.

[5] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench/blob/main/material/MSWE-agent_o4mini_rinja-rs__askama-374.traj.

[6] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench/blob/main/material/RustAgent_clap-rs__clap-2609.traj.

[7] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench/blob/main/material/rustforger_log_bevyengine__bevy-16747.traj.log.

[8] 2025. GitHub Repository. https://github.com/GhabiX/Rust-SWE-Bench/blob/main/material/rustforger_log_tokio-rs__tracing-1017.traj.log.

[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[10] Alibaba Cloud. 2025. Qwen3 API Price. Online. https://www.alibabacloud.com/help/zh/model-studio/models#c5414da58bjgj

[11] Anthropic. 2024. Introducing Claude 3.7 Sonnet. Online. https://www.anthropic.com/news/claude-3-7-sonnet Accessed: 2025-07-15.

[12] Anthropic. 2025. Claude API Documentation. https://docs.anthropic.com/en/home. Accessed: 2025-06-30.

[13] Anthropic. 2025. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family. Accessed: 2025-06-23.

[14] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

[15] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[16] BiomeJS. 2024. Biome: A toolchain for web projects, aimed to provide formatter, linter, and more. https://github.com/biomejs/biome.

[17] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691.

[18] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/3338906.3338947

[19] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:arXiv:2406.01304

[20] Y. W. Chua. 2017. *Appreciating Rust's Memory Safety Guarantees*. https://blog.gds-gov.tech/appreciating-rust-memory-safety-438301fee097

[21] Michael Coblenz, Michelle L Mazurek, and Michael Hicks. 2022. Garbage collection makes rust easier to use: A randomized controlled trial of the bronze garbage collector. In *Proceedings of the 44th International Conference on Software Engineering*. 1021–1032.

[22] Filipe Roseiro Cogo, Xin Xia, and Ahmed E. Hassan. 2023. Assessing the Alignment between the Information Needs of Developers and the Documentation of Programming Languages: A Case Study on Rust. 32, 2, Article 43 (April 2023), 48 pages. https://doi.org/10.1145/3546945

[23] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177* (2023).

[24] The Rust Project Developers. 2024. *The Cargo Book*. https://doc.rust-lang.org/cargo/

[25] The Serde Developers. 2025. The Serialize trait - Serde. Online. https://docs.rs/serde/latest/serde/ser/trait.Serialize.html Accessed: 2025-07-14.

[26] Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1895–1906.

[27] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. 2021. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. 597–616.

[28] Andrew Gallant. 2024. ripgrep: recursively searches directories for a regex pattern. https://github.com/BurntSushi/ripgrep.

[29] Zuchen Gao, Zizheng Zhan, Xianming Li, Erxin Yu, Haotian Zhang, Bin Chen, Yuqun Zhang, and Jing Li. 2025. OASIS: Order-Augmented Strategy for Improved Code Search. *arXiv preprint arXiv:2503.08161* (2025).

[30] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 716–728.

[31] Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. 2025. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. arXiv:2505.08903 [cs.SE] https://arxiv.org/abs/2505.08903

[32] Dana Jansens. 2023. *Supporting the use of Rust in the Chromium project*. https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html Google Security Blog.

[33] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[34] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] https://arxiv.org/abs/2408.02479

[35] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[36] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004* (2023).

[37] Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. 2025. Multi-SWE-bench: 300 curated SWE-bench style tasks from 42 repositories representing 9 programming languages. https://www.swebench.com/multilingual.html. Accessed: 2025-06-28.

[38] killercup. 2024. cargo-edit: A utility for managing Cargo.toml dependencies from the command line. https://github.com/killercup/cargo-edit.

[39] Steve Klabnik and Carol Nichols with contributions from the Rust Community. 2024. *The Rust Programming Language* (2024 ed.). The Rust Project Developers. https://doc.rust-lang.org/stable/book/ Accessed: 2025-07-17.

[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[41] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 579–590.

[42] Abner Li. 2022. *Google reports decline in Android memory safety vulnerabilities as Rust usage grows*. https://9to5google.com/2022/12/01/android-memory-safety-rust/

[43] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. 2024. An Empirical Study of {Rust-for-Linux}: The Success, Dissatisfaction, and Compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 425–443.

[44] Linxi Liang, Jing Gong, Mingwei Liu, Chong Wang, Guangsheng Ou, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. RustEvo$\hat{2}$: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation. arXiv:arXiv:2503.16922

[45] Qingyuan Liang, Zhao Zhang, Zeyu Sun, Zheng Lin, Qi Luo, Xiao Yueyi, Yizhou Chen, Yuqun Zhang, Haotian Zhang, Lu Zhang, et al. 2025. Grammar-based code representation: Is it a worthy pursuit for llms?. In *Findings of the Association for Computational Linguistics: ACL 2025*. 15640–15653.

[46] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091* (2023).

[47] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[48] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. SWE-Lancer: Can Frontier LLMs Earn 1 Million from Real-World Freelance Software Engineering? arXiv:arXiv:2502.12115

[49] Mozilla. 2021. *Mozilla Welcomes the Rust Foundation*. https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/ The Mozilla Blog.

[50] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[51] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2025. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. arXiv:2406.12952 [cs.SE] https://arxiv.org/abs/2406.12952

[52] Yunbo Ni, Yang Feng, Zixi Liu, Runtao Chen, and Baowen Xu. 2024. Towards Fixing Panic Bugs for Real-world Rust Programs. *arXiv e-prints* (2024), arXiv–2408.

[53] OpenAI. 2024. Hello GPT-4o. Online. https://openai.com/index/hello-gpt-4o/ Accessed: 2025-07-15.

[54] OpenAI. 2024. Introducing SWE-bench, verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed on 2025-06-23.

[55] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. Online. https://openai.com/index/introducing-o3-and-o4-mini/ Accessed: 2025-07-15.

[56] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *arXiv preprint arXiv:2411.13990* (2024).

[57] Shane Panter and Nasir Eisty. 2024. Rusty Linux: Advances in Rust for Linux Kernel Development. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3674805.3690756

[58] David Peter. 2024. fd: A simple, fast and user-friendly alternative to 'find'. https://github.com/sharkdp/fd.

[59] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.

[60] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. arXiv:arXiv:2504.08703

[61] Redox OS. 2023. *Redox OS*. https://redox-os.org/

[62] Redox OS Foundation. [n. d.]. Redox OS - The Rust Operating System. https://www.redox-os.org/. Accessed: 2024-06-28.

[63] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232* (2024).

[64] S. Shanker. 2018. *Safe Concurrency with Rust*. http://squidarth.com/rc/rust/2018/06/04/rust-concurrency.html

[65] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. *arXiv preprint arXiv:2412.14234* (2024).

[66] Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. 2025. SK2Decompile: LLM-based Two-Phase Binary Decompilation from Skeleton to Skin. *arXiv preprint arXiv:2509.22114* (2025).

[67] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[68] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286* (2024).

[69] Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. 2025. Decompile-Bench: Million-Scale Binary-Source Function Pairs for Real-World Binary Decompilation. *arXiv preprint arXiv:2505.12668* (2025).

[70] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. A Study on Developer Behaviors for Validating and Repairing LLM-Generated Code Using Eye Tracking and IDE Actions. *arXiv preprint arXiv:2405.16081* (2024).

[71] The Bevy Community. 2024. Bevy: A refreshingly simple data-driven game engine built in Rust. https://github.com/bevyengine/bevy.

[72] The Boa Developers. 2024. Boa: An embeddable and experimental Javascript engine written in Rust. https://github.com/boa-dev/boa.

[73] The Rayon Developers. 2024. Rayon: A data parallelism library for Rust. https://github.com/rayon-rs/rayon.

[74] The Rust Project Developers. 2025. Macros. https://doc.rust-lang.org/book/ch19-06-macros.html. Accessed: 2025-07-15.

[75] The Rust Project Developers. 2025. Module ast. https://doc.rust-lang.org/beta/nightly-rustc/rustc_ast/ast/index.html. Accessed: 2025-07-15.

[76] The Rust Project Developers. 2025. The Rust Programming Language Repository. https://github.com/rust-lang/rust. Source code for the Rust compiler and standard library, including proc_macro and rustc_ast. Accessed: 2025-07-15.

[77] The Rust Team. 2019. *Rust: Empowering everyone to build reliable and efficient software*. https://www.rust-lang.org/

[78] The Rust team. 2021. *The Rust Programming Language*. http://rust-lang.org/

[79] The Rust Team. 2022. *Rust Survey 2021 Results*. https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html

[80] The Servo Project. 2019. *The Servo Browser Engine*. https://servo.org/

[81] The Tokio Authors. 2024. Axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. https://github.com/tokio-rs/axum.

[82] The Tokio Authors. 2024. Tokio: A runtime for writing reliable asynchronous applications with Rust. https://github.com/tokio-rs/tokio.

[83] Arie van Deursen, Maurício Aniche, and Joop Aué. 2016. Delft Students on Software Architecture: DESOSA 2016. *Delft University of Technology* (2016).

[84] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A Survey on Large Language Model Based Autonomous Agents. *Frontiers of Computer Science* 18, 6 (2024), 186345. https://doi.org/10.1007/s11704-024-40231-1

[85] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *International Conference on Machine Learning (ICML)*. https://arxiv.org/abs/2402.01030

[86] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2024. AEGIS: An Agent-based Framework for General Bug Reproduction from Issue Descriptions. arXiv:2411.18015 [cs.SE] https://arxiv.org/abs/2411.18015

[87] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[88] Wikipedia contributors. 2024. Tock (operating system) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Tock_(operating_system) [Online; accessed 28-June-2024].

[89] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).

[90] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. 2024. How far can we go with practical function-level program repair? *arXiv preprint arXiv:2404.12833* (2024).

[91] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.

[92] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[93] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:arXiv:2210.03629

[94] Hongrong Yin, Jinhong Huang, Yao Li, Yunwei Dong, and Tao Zhang. 2025. BugRepro: Enhancing Android Bug Reproduction with Domain-Specific Knowledge Integration. arXiv:2505.14528 [cs.SE] https://arxiv.org/abs/2505.14528

[95] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[96] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. https://doi.org/10.48550/arXiv.2504.02605 arXiv:2504.02605 [cs.SE]

[97] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. 2024. CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios. arXiv:arXiv:2403.19287

[98] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[99] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A Survey on the Memory Mechanism of Large Language Model Based Agents. *arXiv preprint arXiv:2404.13501* (2024). arXiv:2404.13501 https://arxiv.org/abs/2404.13501

[100] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 128–139. https://doi.org/10.1109/ICSE.2019.00030

[101] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 2023. A closer look at the security risks in the rust ecosystem. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–30.