RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code

Pantazis Deligiannis
Microsoft Research, USA
Email: pdeligia@microsoft.com

Akash Lal, Nikita Mehrotra, Rishi Poddar, Aseem Rastogi
Microsoft Research, India
Email: {akashl, nmehrotra, t-ripoddar, aseemr}@microsoft.com

***Abstract*—The Rust programming language, with its safety guarantees, has established itself as a viable choice for low-level systems programming language over the traditional, unsafe alternatives like C/C++. These guarantees come from a strong ownership-based type system, as well as primitive support for features like closures, pattern matching, etc., that make the code more concise and amenable to reasoning. These unique Rust features also pose a steep learning curve for programmers.**

This paper presents a tool called RustAssistant that leverages the emergent capabilities of Large Language Models (LLMs) to automatically suggest fixes for Rust compilation errors. RustAssistant uses a careful combination of prompting techniques as well as iteration between an LLM and the Rust compiler to deliver high accuracy of fixes. RustAssistant is able to achieve an impressive peak accuracy of roughly 74% on real-world compilation errors in popular open-source Rust repositories. We also contribute a dataset of Rust compilation errors to enable further research.

I. Introduction

Code comprehension capabilities of Large Language Models (LLMs) are disrupting the way we build and maintain software systems. LLMs are rapidly becoming an integral part of the workflow, starting from software development [1], to testing [2], repair and debugging [3], [4], [5], [6], [7], [8], [9], [10]. One can interact with pre-trained LLMs [11], [12], [13], [14], [15], without any need of fine-tuning, through *prompts* that details a particular task simply with instructions in natural language. Using prompt engineering to study and harness LLMs has become an active area of research [16], [17].

In this paper, we consider the task of fixing Rust [18] compilation errors using LLMs. Rust, with its safety guarantees, has established itself as a viable choice for low-level systems programming language over the traditional, unsafe alternatives like C/C++. Rust enjoys strong support from both the open-source community [19], [20], and technology companies alike [21], [22].

The Rust typechecker, with a novel *borrow checker* at its core, ensures that Rust programs are free of memory-safety errors and data races that have plagued low-level systems for decades.[1] In addition, Rust has primitive support for features like closures, pattern matching, etc., that make the code more concise and amenable to reasoning.

[1]A Microsoft study found that $\sim$ 70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues [23].

```
15 struct Foo { map: RwLock<HashMap<String, Bar>> }
16
17 impl Foo {
18   pub fn get(&self, key: String) → &Bar {
19     self.map.write().unwrap().entry(key).or_insert(Bar::new())
20   }
21 }
```

Fig. 1: Snippet from a Stack Overflow question

However, this also means that there is a steep learning curve for programmers coming to Rust for the first time. Although Rust tooling (compiler error messages, IDE support [24]) is well-designed to help programmers understand and fix the compilation errors, they can still be intimidating for Rust beginners. A recent survey by the Rust team [25] reports that 83% of the responders who adopted Rust at work found it to be *challenging*. Though adopting a new language and its ecosystem is always challenging, 27% of the responders also say that using Rust is *at times a struggle*.

Consider, for example, Figure 1 showing snippet from a Stack Overflow question [26] about the following error that the Rust compiler emits on this code:

```
error[E0515]: cannot return value referencing temporary value
--> src/example.rs:21:4
   |
19 |   self.map.write().unwrap().entry(key).or_insert(Bar::new())
   |   -------------------------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |   |
   |   returns a value referencing data owned by the current func
   |   temporary value created here
```

For non-experts, making sense of the Rust borrow checking rules, especially with mutex and concurrency can be daunting. And even if the solution is conceptually clear, one still needs to know about Rust libraries that can be used for the fix.

Contributions: We present RustAssistant, an LLM-based tool for automatically fixing Rust compilation errors. RustAssistant specializes prompt construction for the purpose of fixing compilation errors, and uses a novel changelog format in order to harness the LLM capabilities (Section III). RustAssistant is capable of generating non-trivial fixes that are required for real-world scenarios. Section II shows how RustAssistant is able to fix the error described above.

To evaluate RustAssistant, we built a dataset of Rust compilation errors collected from three different sources: (a) 270 micro-benchmarks that we have written ourselves,

covering 270/506 official Rust error codes [27], (b) 50 Rust programs collected from Stack Overflow questions, and (c) 182 GitHub commits with compilation errors from the top-100 Rust crates from `crates.io` (Section IV).

Using RUSTASSISTANT, we systematically study and evaluate capabilities of two LLMs, GPT-3.5 [11] and GPT-4 [12], for fixing Rust compilation errors. With GPT-4, we find that RUSTASSISTANT is able to fix 92.59% of the micro-benchmarks, 72% of the Stack Overflow programs, and 73.63% of the GitHub commits. We also find that GPT-4 performs better than GPT-3.5 in the task. We report on several ablation studies, to investigate the impact of prompting variations (Section V).

We also demonstrate the generalizability of RUSTASSISTANT beyond compilation errors. With only minor prompt modifications, it can handle linter errors generated by Rust Clippy, one of the most popular static analysis tool for Rust [28]. RUSTASSISTANT achieves an accuracy of 75% on the top-10 Rust crates for fixing Clippy-reported errors, which is almost 2.4 times better fix rate than Clippy's own auto-fix feature (RQ4, Section V).

We plan to open-source both our dataset as well as the implementation of RUSTASSISTANT[2].

II. OVERVIEW

This section provides an overview of RUSTASSISTANT and walks through the example from Section I.

Scope: Our goal is to build a toolchain and systematically evaluate the capabilities of LLMs for generating fixes for Rust compilation errors. These fixes must pass the compiler and must also retain the *intended semantics* of the code. The former is an objective criterion while the latter is subjective. Since the code that we start with is not even well-typed, let alone have a well-defined semantics, one requires external judgement to assess the quality of a fix. In the evaluation of RUSTASSISTANT (Section V), we either rely on test cases (fix must build and pass the test) or a comparison with the actual fix made by a developer to establish quality.

We focus on fixing code-related issues and, thus, consider editing only the Rust source files. Errors that require changing a configuration (e.g., adding a package to a `.toml` file) are currently out of scope. We also do not consider errors related to the use of the unsafe keyword, which provides an escape hatch from the typechecker.[3]

Example: The code snippet in Figure 1 contains struct Foo, which maintains a HashMap mapping String keys to Bar values. The hashmap is concurrency-protected with a RwLock, a locking mechanism in Rust that allows multiple readers but at-most one writer at a time. The programmer's intent in the function get is to return a reference to the value mapped to key in the hashmap.

[2]See https://aka.ms/rust-build-fix

[3]We note that the unsafe keyword actually does not turn off all type-checking in Rust. In some cases, especially when interacting with C APIs, an idiomatic translation might necessarily require the use of unsafe. Investigating the performance of LLMs in generating idomatic unsafe code, while interesting, is outside the scope of this paper.

```
ChangeLog:1@src/example.rs
FixDescription: Change the return type of the 'get' method to
return an Arc<Bar> and wrap the Bar in an Arc when inserting
it into the HashMap.
OriginalCode@19–23:
[19] impl Foo {
[20]   pub fn get(&self, key: String) → &Bar {
[21]     self.map.write().unwrap().entry(key).or_insert(Bar::new())
[22]   }
[23] }
FixedCode@19–24:
[19] impl Foo {
[20]   pub fn get(&self, key: String) → std::sync::Arc<Bar> {
[21]     self.map.write().unwrap().entry(key).or_insert_with(
[22]       || std::sync::Arc::new(Bar::new())).clone()
[23]   }
[24] }
```

Fig. 2: Output of RUSTASSISTANT.

```
error[E0308]: mismatched types
--> src/example.rs:22:6
 22 |       || std::sync::Arc::new(Bar::new())).clone()
    |          ^^^^^^^^^^^^^^^^^^^^^^^^^
    |          expected struct 'Bar', found struct 'Arc'
```

Fig. 3: Error after the first fix suggested by RUSTASSISTANT

The implementation of get first calls RwLock::write to get exclusive write access to the hashmap object. The write access is released as the guard goes out-of-scope, e.g., when the function call returns. The function then proceeds to read the value of key, and return a reference to the read value. When this code is compiled, the compiler complains with the error shown in Section I.

Rust maintains a list of all the error codes that can be emitted by the compiler [27]. Here the error code is E0515 on line 21, meaning that the function is trying to return reference to a local variable. The error comes from the Rust borrow checker. The returned reference, which is derived from the mutex owned hashmap, escapes the function scope, and therefore, outlives the mutex guard lifetime—a violation of the borrow checking rules.

Let's see how RUSTASSISTANT fixes the code. RUSTASSISTANT first invokes the Rust compiler on the input code and collects the error message. It then feeds the code and the error message, along with instructions in a prompt, to the LLM. RUSTASSISTANT is parametric in the choice of LLM; we show the interactions with GPT-4 in this section. Figure 2 shows the output of GPT-4. The output contains the suggested fix in text and a code patch in the form of a *changelog*; this format is explained in the next section.

The suggested fix is to change the return type of get to Arc<Bar>, and also insert Arc<Bar> in the hashmap. Arc in Rust is an atomically reference-counted, thread-safe pointer. With this change, the get function can return a copy of the value, an Arc pointer that points to the same heap location as the value in the map (clone creates the copy).

RUSTASSISTANT parses this LLM output, applies the suggested patch, and compiles the program again. This time, the

Rust compiler complains with the error shown in Figure 3. The error is about the mismatch between the declared type of hashmap, mapping String keys to Bar values, and the usage of it as a map from String keys to Arc<Bar> values—indeed, the previous patch did not fix the declaration of the hashmap.

RUSTASSISTANT sends the code and the error to the LLM again. In this instance, GPT-4 responds with the following fix, correctly suggesting to change the type of map.

```
ChangeLog:1@src/example.rs
FixDescription: Change the type of values stored in the HashMap
    to Arc<Bar>.
OriginalCode@16–16:
[16]   map: RwLock<HashMap<String, Bar>>
FixedCode@16–16:
[16]   map: RwLock<HashMap<String, std::sync::Arc<Bar>>
```

After applying this patch, the compilation succeeds and RUSTASSISTANT returns. Using `Arc` is also the accepted Stack Overflow answer for this question [26].

III. RUSTASSISTANT IMPLEMENTATION

Algorithm 1 The RUSTASSISTANT algorithm.

Require: m: LLM, N: Number of completions
Require: $project$: Rust project
1: $errs \leftarrow$ check($project$)
2: **while** $errs \neq \emptyset$ **do**
3: $e \leftarrow$ choose_any($errs$)
4: $g \leftarrow \{e\}$
5: $snap \leftarrow project$
6: **while** $g \neq \emptyset$ **do**
7: $e' \leftarrow$ choose_any(g)
8: $p \leftarrow$ instantiate_prompt(e')
9: $n \leftarrow$ invoke_llm(m, p, N)
10: $c \leftarrow$ best_completion($project, n$)
11: $project \leftarrow$ apply_patch($project, c$)
12: $g \leftarrow$ check($project$) $-$ $errs$
13: **if** giveup() **then**
14: $project \leftarrow snap$
15: **break**
16: **end if**
17: **end while**
18: $errs \leftarrow$ check($project$)
19: **end while**

RUSTASSISTANT is a command-line tool that takes as input the filesystem path to a Rust project, potentially with errors. RUSTASSISTANT parses the project to build an in-memory index of the Rust source files. The index allows RUSTASSISTANT to retrieve the contents of the files, edit them, or even revert them to a previous state.

RUSTASSISTANT must handle the complexities of fixing errors in real-world scenarios. Source files can be large relative to the LLM prompt sizes that were available to us (maximum of 32K tokens, for GPT-4), and most of the code in a file might not be relevant to a reported error any way. RUSTASSISTANT, therefore, performs *localization* for each error to identify relevant parts of the source code and presents only those parts to the LLM. This implies that we need a way of parsing the LLM response to know which change needs to be applied where. We tried a naive approach where we asked the LLM to simply give us the entire revised code snippets but this did not work well (Section V). We instead define a simple, but effective, *changelog* format that only captures the *changes* that need to be made to the given code snippets. We describe this format in the prompt and instruct the model to follow it. RUSTASSISTANT uses a lightweight parser to understand the changelog in the LLM's response and can then easily apply the changes to the original source code. This approach significantly increases RUSTASSISTANT's accuracy, possibly because the LLM output stays focussed on the code changes. The format is also easy to parse and check for validity.

Algorithm 1 shows the RUSTASSISTANT core algorithm. The algorithm starts by invoking the compiler on the project to gather the initial list of errors, and starts fixing them one at a time (line 2).

*a) **Inner loop for fixing an input error:*** The inner loop (lines $6-17$) iterates with the LLM with the goal of fixing a single input error (e). During this iteration, the source files may change and those changes may themselves induce additional errors. To accommodate this, we introduce an abstraction called an *error group* as the working unit of the RUSTASSISTANT inner loop. An error group is a set of errors that RUSTASSISTANT is currently trying to fix. An error group is initialized with the input error (line 4) and may grow or shrink within the loop. The loop terminates when either the error group becomes empty, implying that the original error e was fixed, or RUSTASSISTANT gives up on the error group (line 13), in which case the project is restored to its initial state at the beginning of the outer loop. We now explain the body of the inner loop.

*b) **Prompt construction (instantiate_prompt):*** For each error (e') in the current error group, RUSTASSISTANT constructs a prompt p, shown in Figure 4 (the headings are for illustration purposes only), asking for a fix to the error. The prompt is parameterized over error-specific content, using the `{}` syntax. The `preamble` section instantiates the compiler command that was used (`cmd`). The next section of the prompt contains the text of the error message. This is followed by code snippet(s) that RUSTASSISTANT deems necessary to present to the LLM for fixing the error. These are obtained by first identifying source locations in the error. The Rust compiler, for instance, not just points to the error location, but to related locations as well. In the example error message below, the location after note is a related location:

```
error[E0369]: binary operation '>=' cannot be applied to '
    Verbosity'
--> src/logger.rs:53:21
 53 |     if self.verbosity >= Verbosity::Exhaustive {
    |        -------------- ^^^ ---------------------
note: an implementation of 'PartialOrd<_>' might be missing
--> src/logger.rs:16:1
 16 | pub enum Verbosity {
    | ^^^^^^^^^^^^^^^^^^ must implement 'PartialOrd<_>'
help: consider annotating with '#[derive(PartialEq, PartialOrd)]'
```

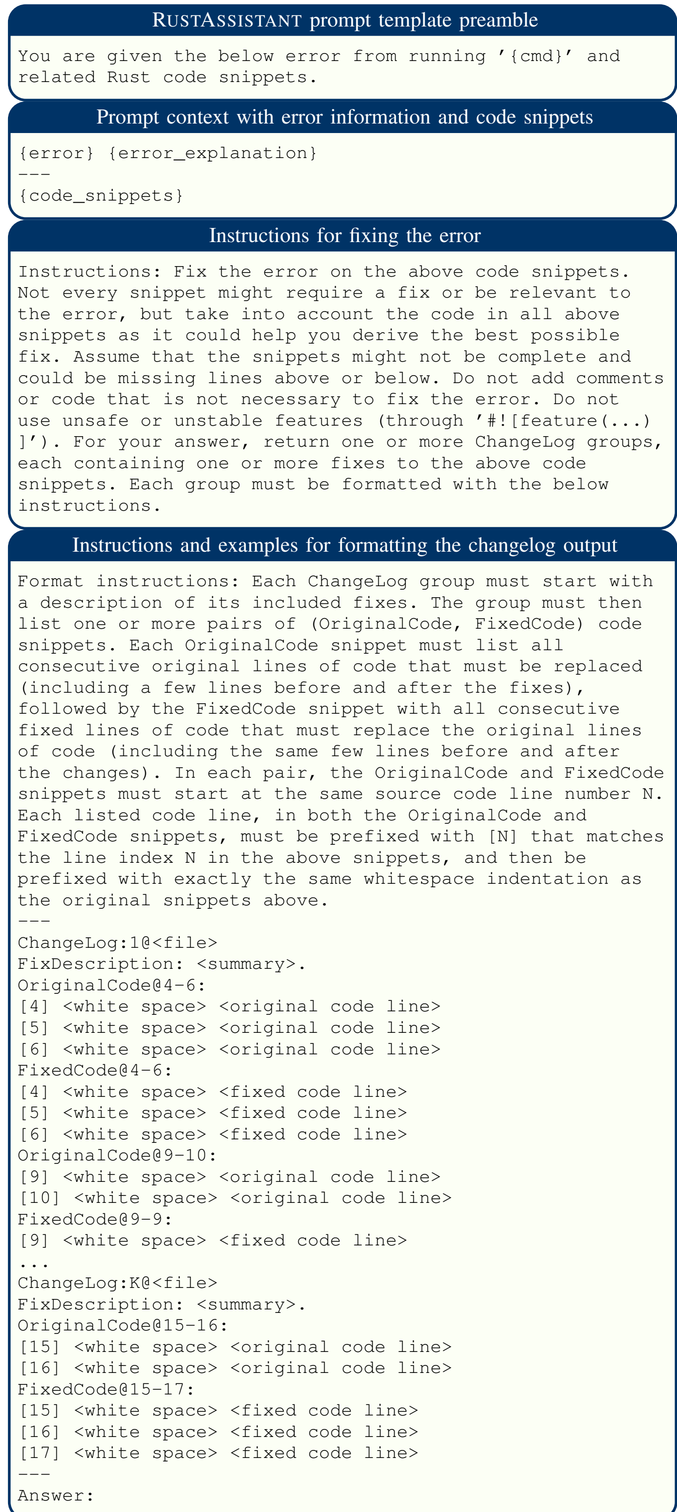

Fig. 4: The RustAssistant prompt template.

RustAssistant then extracts a window of ±50 lines around each location, and adds these snippets to the prompt. (The size of this window is configurable; we settled on ±50 to balance prompt size against accuracy on a small subset of benchmarks.) RustAssistant also adds the line number for each line of code as a prefix, which helps the LLM to better identify the code lines in the prompt. In an initial attempt, we tried only extracting code segments in a proper lexical scope (e.g., the entire body of a function where a relevant line appears). This not only increased the complexity of our tooling (because one needs to parse the Rust code and obtain an AST) but we also found that LLMs are robust even to non-lexical scopes. We decided in favor of keeping our tooling simple.

The next section of the prompt (`instructions`) are simple instructions that ask for a fix. For instance, it instructs the model to avoid adding *unsafe* code, in an effort to keep the tool focused on generating good Rust code.

The final section of the prompt contains instructions to the LLM for formatting the output, as a list of one or more *change logs*. It contains meta-instances of change log to explain the format to LLM. Each changelog begins with an ID numbered starting with 1 and the source file to which it is applied (`ChangeLog` line in Figure 4). The next part (`FixDescription` line) asks the LLM to add a free-form description of the fix that it is proposing. This description is not even parsed by RustAssistant. Its purpose is to enable *chain-of-thought* reasoning [29] that has been found to help increase the accuracy of the model's response. Next part is a repetition of the input code that was given to the model (`OriginalCode`). This part is *defensive* because it is a repetition of the input; RustAssistant rejects the changelog if the `OriginalCode` segment fails to match the actual original code. Finally, the output must have the fixed code (`FixedCode`) that should replace all the lines of the original code. If this segment is empty, for example, then it implies that the corresponding original code segment should be deleted. There are other defensive checks in the changelog format: each of `OriginalCode` and `FixedCode` segments must mention the line number range; and this number range repeats again in the code segment. All such checks act as a guard; change logs are rejected when this information fails to match.

*c) **LLM invocation (invoke_llm)**:* Once the prompt is instantiated, RustAssistant invokes the LLM with the prompt (line 9) asking for N *completions*, essentially, N responses to the same prompt. On receiving these completions, RustAssistant ranks them and picks the best completion (line 10). To rank the completions, RustAssistant applies all the changelogs in a completion and counts the number of resulting errors reported by the compiler. The completion that results in the least number of errors is ranked the highest.

The best completion is applied to the project (line 11), the current error group is updated (line 12) and RustAssistant then continues with the inner loop. When the inner loop completes, RustAssistant updates the set of pending errors (line 18) because it is possible that fixing one error group caused the errors to change. (As a detail, errors that were previously given up, on line 13, are not tried again; but this is omitted from the algorithm).

RustAssistant uses a few heuristics to ensure termination

of the inner and the outer loops. First, it provides a configurable option (default 100) to limit the maximum number of unique errors that an error group can have over its lifetime in the inner loop. If this limit is reached, the inner loop gives up. Second, if the set of errors in an error group does not change across iterations of the inner loop, RUSTASSISTANT considers it as not making progress and gives up on the error group. The outer loop is bounded to run for as many iterations as the initial number of errors obtained on line 1. (For the purpose of checking if two errors are same, which is needed when performing set operations, RUSTASSISTANT represents an error as the concatenation of its error code, error message, and the file name, without any line numbers.)

IV. RUST ERROR DATASET

We build a dataset of Rust compilation errors collected from three different sources. We also compile a collection of linting errors reported by Clippy on these sources.

A. Micro-benchmarks

Rust offers a comprehensive catalog of errors, indexed by error codes, that the Rust compiler may emit. The catalog is accompanied by small programs that trigger the specific error codes [27]. To build our micro-benchmarks dataset, we wrote small Rust programs, one per error code, designed specially to trigger that error code. We wrote these programs ourselves, using the snippets in the Rust catalog as a reference.

We consider 270 error codes out of a total of 506. We exclude error codes that are no longer relevant in the latest version of the Rust compiler (1.67.1). Additionally, we exclude all errors related to package use, build configuration files, and foreign function interop, as well as error codes on the use of unsafe; as mentioned in Section II, these errors are out-of-scope for us. We also create a unit test for each error code. This test is not shown to RUSTASSISTANT; it is written in a separate file. It is only used to check if the fix meets the intended semantics of the program. As an example, following is a snippet from our micro-benchmark for error code E0382 (borrow of moved value), with the corresponding testcase (comments are not present in the benchmark):

```
pub fn get_value() -> u32 {
    let calc1 = Calculator { val: 6 };
    let mut calc2 = calc1; calc2.val = 5;
    // inc increments val by 1
    <Calculator as Math>::inc(&calc1)
    // above line borrows calc1 after it has been moved (illegal)
}
#[test]
fn test_e0382() {
  let result = get_value(); assert_eq!(result, 6);
}
```

We further classified each of the errors codes into one of the six categories: Syntax, Type, Generics, Traits, Ownership, and Lifetime. The primary objective of this benchmark is to determine if RUSTASSISTANT is more proficient at fixing certain types of errors compared to others.

B. Stack Overflow (SO) code snippets

Stack Overflow (SO) is a popular online community where programmers and developers seek help for coding issues. We manually scrape SO to collect questions about Rust compilation errors. To limit our effort, we concentrate on memory-safety (including ownership and lifetime) and thread-safety issues, two areas in which the Rust type system is stricter than C/C++.

To ensure that the questions are relevant and substantial, we apply some filtering criteria. For example, we require each question to have at least one answer, we exclude cases that we deem trivial (e.g., the question is misclassified or contains syntax errors unrelated to the intended category), as well as exclude questions that were tagged as duplicates of a previously posted question. After applying these filtering criteria, we select the first 50 most relevant questions with total 65 compilation errors. 94% of the errors are related to the Rust-specific concepts of lifetime, ownership, and traits.

Code snippets in these questions are not always self-contained. We manually add code and stubs to scope the compilation issue to only what was asked in the corresponding SO question. This process of completing the program snippet was generally quite straightforward. We manually add test cases also, as we did for the micro-benchmarks. For example, for the code snippet shown in Section I, we wrote the following test:

```
pub fn get_value() -> usize {
    let foo = Foo { map: RwLock::new(HashMap::new()) };
    let bar = foo.get("key".to_string());
    bar.val.load(Ordering::SeqCst)
}
#[test]
fn test_so042() {
  // 0 is the default value that is added for a new key
  let result = get_value(); assert_eq!(result, 0);
}
```

C. Top-100 crates

For a more comprehensive real-world evaluation, we look at the GitHub repositories of the top-100 Rust crates (the most widely-used Rust library packages) from `crates.io` [30]. We examine the history of these repositories and identify commits that have compilation errors (we clone the commits and build them locally, we also filter out commits where the errors are out-of-scope). We found 182 such commits. The benchmark then is to fix the commits so that they pass the Rust compiler. In our evaluation, we manually audit the fixes to check whether they preserve the intended semantics (see RQ3 in Section V). These contain 204 errors, out of which 61 errors are related to ownership, lifetime, and traits.

D. Linting errors

To build a dataset of linting errors, we pick top-10 crates, and run `rust-clippy` on the latest commit in the main branch of their corresponding GitHub repositories. Clippy [28] is one of the most popular open-source static analysis tool for Rust with roughly 10K stars on GitHub. It is designed to help

developers write idiomatic, efficient, and bug-free Rust code by providing a set of predefined linting rules. Clippy also provides helpful messages and suggestions to guide developers in making improvements to their code. Fixing Clippy errors tests the ability of RUSTASSISTANT to generalize beyond compilation errors. Our dataset has a total of 346 Clippy errors.

Clippy has multiple categories of checks [28]. For our dataset, we only consider Pedantic, Complexity, and Style. The rest of the categories did not raise errors in the top-10 crates. Additionally, there is a category called Nursery, but it consists of lints that are not yet stable, so we exclude it from our consideration. Pedantic refers to stylistic or convention violations, Complexity refers to unnecessarily complex or convoluted code that hamper maintainability, and Style covers various linting rules related to code style and best practices, focusing on conventions such as naming, spacing, formatting, and other stylistic aspects of the code.

V. EVALUATION

We evaluate RUSTASSISTANT to answer the following research questions:

1) **RQ1:** To what extent is RUSTASSISTANT successful in fixing Rust compilation errors?
2) **RQ2:** How effective are different prompting strategies and algorithmic variations?
3) **RQ3:** How accurate are the fixes generated by RUSTASSISTANT for real-world repositories?
4) **RQ4:** Can RUSTASSISTANT generalize to fix errors reported by a Rust static analyzer?

LLM Configuration: We adopted a deterministic approach by using `top_p=1`, meaning that the most likely token is selected at each generation step. To maintain the focus and consistency of the outputs, we opt for a low temperature of 0.2. While the maximum length of the generated output is set to the default value of 800 tokens, in practice, our changelog snippets are concise and significantly smaller in length.

We evaluate both GPT-3.5-turbo [11] (which we call as GPT-3.5) and GPT-4 [12]; a comparison between them helps understand the effect of model scaling on RUSTASSISTANT's accuracy. For both LLMs, we also vary the number of LLM completions (#N) from 1 to 5 to provide insights into the optimal balance between computation time and quality of the fixes.

A. RQ1: To what extent is RUSTASSISTANT successful in fixing Rust compilation errors?

For micro-benchmarks and SO, we say that a benchmark is *fixed* by RUSTASSISTANT if the result code compiles and passes its unit test. A generated fix can fail for three reasons: (1) **Format Errors**, where the generated changelog is not correctly formatted, or the format is correct but the original code or lines do not match, leading to the rejection of the changelog; (2) **Build Errors**, the fix when applied still results in compilation errors; (3) **Failed Tests**, there are no formatting or compilation errors, but the corresponding unit test fails. For the top-100 crates benchmark, we report the

				# Failures		
	Model	N	#Fixed	Fmt	Build	Test
`Micro-benchmarks` (`cargo fix`: 25 / 270)	`GPT-3.5`	1	143	59	56	12
		5	199	44	10	17
	`GPT-4`	1	249	✗	8	13
		5	**252**	✗	4	14
`Stack Overflow` (`cargo fix: 1/50`)	`GPT-3.5`	1	12	18	18	2
		5	25	21	2	2
	`GPT-4`	1	**37**	1	7	5
		5	36	3	4	7

TABLE I: Evaluation on micro-benchmark and SO datasets (N is the number of completions, Fixed rate is out of 270 for micro-benchmark and 50 for Stack Overflow).

number of commits that successfully compile after running our tool as well as the number of compilation errors fixed across all the commits. As an estimate for the cost of running RUSTASSISTANT, we also report per commit average of (a) number of LLM queries, (b) number of input and output tokens per prompt, and (c) running time.

As a point of comparison, for each dataset, we report the number of benchmarks that can be fixed by `cargo fix -broken-code`, a Rust command that tries to fix compilation errors by applying any suggestions from the Rust compiler [31]. This tool is an open source effort maintained by the Rust community and serves as a pattern-based baseline in our evaluation, i.e., fixes that can be implemented as a pattern derived from a compilation error.

1) Micro-benchmarks: The top half of Table I shows the performance of RUSTASSISTANT on micro-benchmarks. RUSTASSISTANT achieves a peak accuracy of 93%. In comparison, `cargo fix` is only able to fix less than 10% of the errors. GPT-4's performance is significantly better than GPT-3.5, so model scaling helps. Increasing the number of completions helps, but only minimally for GPT-4, potentially because its fix rate is already very high with $N = 1$. We also notice that GPT-3.5 often fails to follow our formatting requirement in its output (59 failures with $N = 1$) or fails to satisfy the compiler (56 failures with $N = 1$).

Interestingly, sometimes RUSTASSISTANT produces a fix that fails the corresponding unit test (e.g., with GPT-4, there are 13 test failures with $N = 1$). Consider the code alongside. It fails to build because it uses the bitwise operator <<= on a floating point value (error code E0368). GPT-4 proposes a fix to change the operator to multiplication, but it keeps the unit as 2.0, instead of changing it to 4.0. Expectedly, this fix fails the unit test that we wrote for this benchmark.

```
pub fn get_value()
        → f64 {
    let mut val: f64
        = 7.0;
    val <<= 2.0;
    val
}
```

2) Stack Overflow (SO) benchmarks: The bottom half of Table I shows the results on the SO benchmarks. Overall trends, with respect to the two models and the number of completions, are similar to microbenchmarks. However, the fix

Model	#N	#Commits	#Errors	Avg Prompts / Commit
GPT-3.5	1	55 / 182	414 / 925	11
	5	65 / 182	509 / 925	12
GPT-4	1	99 / 182	796 / 925	10
	5	**134 / 182**	**846 /925**	10

TABLE II: Evaluation on the top-100 Rust crates (`cargo fix` fixes 1/182).

percentages are consistently lower. RUSTASSISTANT is able to achieve a peak fix percentage of 74%. `cargo fix` shows a more drastic drop in performance, it is able to fix only 1/50. As the example from Section II shows, fixes can require the introduction of new concepts and types (such as reference counting via Arc) that are hard to automate in a pattern-based manner from the compilation error message.

Across the micro-benchmarks and SO benchmarks, we manually investigate the reasons for failure. In some cases, the model suggests a correct partial fix but then does not follow up with the additional fixes required. In other cases, it gets stuck in a loop where it proposes a fix and undoes it in the next iteration, causing RUSTASSISTANT to give up. In a few cases, the model tries to import a package that it needs, but RUSTASSISTANT is not prepared to edit the `.toml` project file for actually doing the import. It is possible that further refinement of the LLM prompt can fix such issues; we leave it for future work.

For the successful benchmarks, with GPT-4, RUSTASSISTANT required up to 6 iterations of the inner-loop (Algorithm 1) to come up with a fix. In the case of SO benchmarks, this number goes up to 15.

3) Top-100 crates benchmark: Table II presents results on the top-100 crates benchmark. RUSTASSISTANT is able to achieve an impressive peak accuracy of 91.46% in terms of fixing errors, matching what is also observed in the micro-benchmarks. When we consider the ability to fix all errors in a commit, the fix rate is lower, but still impressive at 73.63%, i.e., roughly three-fourths of the commits could have been automatically fixed by RUSTASSISTANT.

As Table II shows, RUSTASSISTANT requires on-average 10 round-trip interactions with GPT-4 to fix an entire commit. For the configuration with 5 completions, we further computed the token costs. RUSTASSISTANT used, on average, 2751 input tokens per prompt and 679 output tokens per prompt. With current OpenAI pricing, this amounts to only USD 1.2 to fix all errors in a commit.[4] The price per commit with GPT-3.5 is much lower (USD 0.07) at the expense of lower fix quality.

In terms of running times, RUSTASSISTANT spends majority of the time either building the code or querying the LLM. The latter is the dominant cost, especially because our OpenAI endpoints were throttled, causing a majority of our requests to timeout. For fair accounting, and because RUSTASSISTANT runs sequentially, we remove requests that timed out from consideration. Then RUSTASSISTANT spends 22 seconds on average per commit on building the code, and 249.9 seconds waiting on GPT-4. It should be possible to improve this running time by parallelizing RUSTASSISTANT (across different errors) but we leave this for future work.

B. *Qualitative analysis of fixes generated by* RUSTASSISTANT

We further analyzed the fixes generated by RUSTASSISTANT to investigate its ability to fix non-trivial errors. We concentrate on compilation errors related to ownership, lifetime, and traits, arguably the concepts that Rust programmers struggle with most. We identify 23 Rust error codes that are related, and refer to this set as $\mathcal{S}$.[5]

Among micro-benchmarks, RUSTASSISTANT is able to fix 90/99 errors from $\mathcal{S}$. In the SO dataset, there are 43/50 programs that have at least one compilation error from $\mathcal{S}$; RUSTASSISTANT is able to fix 31 of them. Finally, in the top-100 crates benchmark, 60/182 commits have at least one compilation error from $\mathcal{S}$, out of which RUSTASSISTANT fixes 58 in a *runtime behavior preserving* way (Section V-D describes this criteria more precisely).

We now demonstrate, using examples, that the fixes do not follow from the compiler error message and that they require understanding the code that needs fixing. Consider the following error from parking_lot, one of the crates in our top-100 dataset:

```
error[E0713]: borrow may still be in use when destructor runs
  --> src/mutex.rs:318:23
    |
286 |   impl<'a, T: ?Sized + 'a> MutexGuard<'a, T> {
    |        -- lifetime ''a' defined here
...
317 | /        MutexGuard {
318 | |            borrow: f(orig.borrow),
    | |                      ^^^^^^^^^^^
319 | |            raw: orig.raw,
320 | |            marker: orig.marker,
321 | |        }
    | |_________- returning this value requires that '*orig.borrow' is borrowed for ''a'
322 |      }
```

Following is some context relevant to the error; the error above is inside the map function.

```
pub struct MutexGuard<'a, T: ?Sized + 'a> { ... borrow: &'a mut T, }

pub fn map<U: ?Sized, F>(orig: Self, f: F) -> MutexGuard<'a, U> where F: FnOnce(&mut T) -> &mut U { ... }
```

The error occurs because the mutable borrow of orig.borrow at line 318 (indicated in the error message) conflicts with the destructor of orig invoked at the end of map. Since map takes the ownership of the orig argument, the Rust compiler adds this destructor call. RUSTASSISTANT suggests a fix that wraps orig in std::mem::ManuallyDrop inside map to inhibit compiler from calling its destructor, and adjusts the uses of orig to account for it (the actual fix in the repository uses std::mem::forget which achieves the same effect).

[4] See https://openai.com/pricing; prices can vary with time.

[5] E0106, E0311, E0515, E0621, E0700, E0726, E0382, E0594, E0499, E0502, E0505, E0507, E0521, E0596, E0597, E0716, E0119, E0223, E0271, E0277, E0405, E0445, E0782

This fix is non-trivial. The official documentation of this error code (E0713) suggests taking the function argument by reference. However, RUSTASSISTANT suggests an alternate fix that is closer to the developer intent.

We also observe that RUSTASSISTANT is able to suggest fixes that are *code-dependent*, i.e., different fixes for the same compilation error code. For example, one of the SO benchmarks has the following error (the for loop is the relevant code):

```
for (i, m) in self.margins.iter_mut().enumerate() {
    m.y += 1.;
    if m.y > 640. { self.margins.remove(i); }
}
error[E0499]: cannot borrow `self.margins` as mutable more than
      once at a time
--> src/example.rs:29:17
   |
26 |        for (i, m) in self.margins.iter_mut().enumerate() {
   |                      ---------------------------------
   |                      |
   |                      first mutable borrow occurs here
   |                      first borrow later used here
...
29 |                self.margins.remove(i);
   |                ^^^^^^^^^^^^^^^^ second mutable borrow occurs here
```

The error complains about two mutable borrows of self. margins, a std::Vec. RUSTASSISTANT fixes the error by converting the for loop into a while loop that uses an index variable for iteration. With this fix, there is only one mutable borrow of self.margins.

```
let mut i = 0;
while i < self.margins.len() {
    self.margins[i].y += 1.;
    if self.margins[i].y > 640. { self.margins.remove(i); }
    else { i += 1; }
}
```

While for the same error code in another SO benchmark:

```
struct Pool { strings: Vec<String> }
impl Pool {
    pub fn some_f(&mut self) -> Vec<&str> {
        let mut v = vec![];
        for i in 1..10 {
            let string = format!("{}", i);
            let string_ref = self.new_string(string);
            v.push(string_ref);
        }
        v
    }

    fn new_string(&mut self, string: String) -> &str { ... }
}
error[E0499]: cannot borrow `*self` as mutable more than once at
      a time
--> src/example.rs:19:30
   |
14 |   pub fn some_f(&mut self) -> Vec<&str> {
   |              - let's call the lifetime of this reference '1'
...
19 |     let string_ref = self.new_string(string);
   |                      ^^^^^^^^^^^^^^^^ `*self` was mutably borrowed
      here in the previous iteration of the loop
...
23 |     v
   |     - returning this value requires that `*self` is borrowed for
      '1'
```

Model	Prompt	G	#N	#Commits	#Errors
GPT-3.5	P0	✓	1	18 / 182	298 / 925
			5	16 / 182	382 / 925
GPT-3.5	P4	✗	1	18 / 182	46 / 925
			5	59 / 182	129 / 925

TABLE III: Ablations on the top-100 Rust packages.

RUSTASSISTANT fixes it by changing the return type of some_f to Vec<String> and updating the new_string function to return a String instead of a reference. This example illustrates the diversity of potential fixes for the same error codes.

C. RQ2: How effective are different prompting strategies and algorithmic variations?

We perform an ablation study by permuting between different prompting and algorithmic variations, in order to identify the most effective features of RUSTASSISTANT. We consider five prompt variants, which differ in the way RUSTASSISTANT asks LLMs to output the fixes, i.e. the output formatting instructions.

1) **P0 (Basic)**: This variant serves as the baseline. It does not have the changelog section; it instead asks for the complete revised snippets.
2) **P1 (ChangeLog-basic)**: This uses the changelog, but only the `FixedCode` section, without the line number prefixes.
3) **P2 (Line-prefixes)**: In addition to P1, we require line number prefixes in front of code snippets.
4) **P3 (Localization)**: In addition to P2, we require the original code section.
5) **P4 (Description-first)**: This is the full prompt of Figure 4, i.e., P3 with the `FixDescription` section.

For algorithmic ablations, we vary the number of completions (#N) to either 1 or 5 (already reported for RQ1), and we turn off error grouping. Without error grouping, the RUSTASSISTANT algorithm has a single loop that attempts to fix one error at a time from the current bag of errors.

Table III shows the results on the top-100 crates benchmark. We see that P0 results in very poor performance (compare its first two rows with those of Table II). We found that the model, when returning the fixed code snippet would get tempted in making code changes that were unrelated to the task of fixing the compiler error. This justifies the need for investing in changelog format to keep the model focused on the fix.

Table III also shows that turning off grouping significantly drops the error fix rate (compare the last two rows of Table III with the first two rows of Table II). Without grouping, RUSTASSISTANT would fix errors in a random order, which increased the chances of it getting stuck with an error that it could not fix, leading to an ever-increasing blow-up of code changes and resulting errors. Error grouping helps in detecting such cases, allowing RUSTASSISTANT to gracefully give up on them, and then move on to the other errors in the project. This justifies the importance of error grouping.

	Prompt Variant	#Fixed	#Failures Format	Build	Test
P1	ChangeLog-basic	139 / 270	120	3	8
P2	Line-prefixes	196 / 270	61	3	10
P3	Localization	247 / 270	0	6	17
P4	Description-first	252 / 270	0	4	14

TABLE IV: Evaluating variants of the RUSTASSISTANT prompt template on micro-benchmarks with GPT-4.

	# Commits
Unambiguous	55
Matching	41
Non-matching, same runtime behaviour	29
Different runtime behavior	9

TABLE V: Analysis of fixed commits in top-100 benchmark

Table IV presents the results for prompt ablations with GPT-4. It demonstrates that the addition of each new feature to the changelog format raises accuracy by a significant margin. The basic format (P1) only provides roughly 51% accuracy. We saw that P1 response would trip most on the formatting of its output, an important requirement in order to handle large code bases. The number of formatting errors reduce significantly as the changelog format is improved. It is interesting that the simple act of describing the fix (going from P3 to P4) helps the model accuracy.

D. RQ3: How accurate are the fixes generated by RUSTASSISTANT *for real-world repositories?*

To answer RQ3, we qualitatively examine the fixes generated by RUSTASSISTANT on all the 134 fixed commits from the top-100 crates benchmark. As shown in Table V, we categorize the commits into 4 categories. The categories are defined in term of individual error fixes, and a commit is classified into a category if all its fixes belong to the category or to the categories above it.[6] For example, a commit is classified as *Non-matching, same runtime behavior* if all its fixes are either *Unambiguous*, or *Matching*, or *Non-matching, same runtime behavior*.

Unambiguous fixes classify fixes for errors like syntax errors (e.g., missing ; or braces), missing instantiations for generics type parameters, type-incorrect format string specifiers, etc. Since there is mostly a unique way to fix these errors, and the RUSTASSISTANT fix passes the Rust typechecker, we consider it good.

Matching fixes classify the fixes where the fix matches the developer fix in the repository—we checked this by comparing the patched code from RUSTASSISTANT with the corresponding code from the latest commit in the main branch of the repository. This category contains non-trivial examples that provide some evidence that the LLMs have learnt common Rust idioms. For example, a common pattern to destruct a pointer in Rust is to wrap it using Box::from_raw(ptr), and let the Box destructor call the destructor for the pointer. In a few instances in our benchmark, void returning functions had Box::from_raw(ptr) as the last statement, which the Rust compiler complains about as unused value. RUSTASSISTANT fixes these errors by rewriting it as drop(Box::from_raw(ptr)), which matches the actual fix in the repository.

Similarly, there are examples where the Rust compiler complains about a function modifying an argument through an immutable reference. RUSTASSISTANT fixed such cases by changing the signature of the function to demand &mut references. Another class of errors in this category are deprecation warnings/errors, e.g.:

```
error: use of deprecated associated function
'core::sync::atomic::AtomicPtr::<T>::compare_and_swap':
Use 'compare_exchange' or 'compare_exchange_weak' instead
```

The fix produced by RUSTASSISTANT follows the suggestion in the error, and comes up with the following patch:

```
- atom.compare_and_swap(ptr as _, shared as _, Ordering::
      AcqRel);
+ atom.compare_exchange(ptr as _, shared as _, Ordering::
      AcqRel, Ordering::Acquire).unwrap_or_else(|x| x);
```

The third category, non-matching but same runtime behavior, classifies the fixes where the fix produced by RUSTASSISTANT does not match the fix in the repository, but to the best-of-our estimation, the fix does not alter the runtime behavior of the program (recall that the fix passes the Rust typechecker). This is an interesting category where the LLM produces reasonable fixes that do not match the programmer intent. One example in this category is the following error:

```
error[E0015]: cannot call non-const fn 'ArrayString::<CAP>::
      capacity' in constant functions
```

The error is in a function defined as const:

```
pub const fn remaining_capacity(&self) -> usize
```

The compiler complains that the function calls another function capacity, passing it the self argument, but capacity is a non-const function. LLM chose to fix this error by removing the const qualifier from remaining_capacity, whereas the programmer fixed it by adding const qualifier to capacity. We found similar instances related to other qualifiers such as mut and public.

Another example in the category is the error in the following trait definition:

```
pub trait MendSlice { fn mend(Self, Self) -> Result<Self,(Self,Self)
      >; }
```

Rust compiler complains that it needs to statically know the size of Self, and suggests bounding Self with the Sized trait:

```
error[E0277]: the size for values of type 'Self' cannot be known at
      compilation time
      |
151 | fn mend(Self, Self) -> Result<Self, (Self, Self)>;
      | ^^^^^^^^^^^^ doesn't have a size known at compile-time
      |
      = note: only the last element of a tuple may have a
            dynamically sized type
 help: consider further restricting 'Self'
```

[6]Note that a commit can have multiple errors and hence multiple fixes.

Category	Clippy Fix%	Clippy #Fixed	RustAssistant Fix%	RustAssistant #Fixed
Complexity	50.00%	17 / 34	**91.18%**	**31 / 34**
Pedantic	26.12%	76 / 291	**71.48%**	**208 / 291**
Style	76.19%	16 / 21	**95.24%**	**20 / 21**
Total	31.50%	109 / 346	**74.86%**	**259 / 346**

TABLE VI: Evaluating RustAssistant (GPT-4) against Clippy's auto-fix on the top-10 Rust packages.

```
    |
151 | fn mend(Self, Self) → Result<Self, (Self, Self)> where Self:
      Sized;
```

The RustAssistant fix in this case was to bound the Self argument at the level of MendSlice definition

```
pub trait MendSlice: {Sized} { fn mend(Self, Self) → Result<Self,(
      Self,Self)>; }
```

whereas in the repository, the fix is what the error message suggested. This is an interesting case since the LLM chose to ignore the suggestion in the error message.

The final category is for the fixes where the changes introduced by the LLM, again to the best of our estimation, alter the runtime behavior of the program. In a few of these cases, the LLM patch was blatantly wrong, e.g., it removed some code, introduced alternate implementations of some unrelated functions, etc. However, in some cases it was understandable that the LLM patch did not match the developer. An example is as follows. Rust supports enumerated types and a match construct to inspect the variant of the enum and execute different code based on the variant. If the match is not exhaustive, i.e. it doesn't mention all the variant cases, the compiler raises an error. In a few cases of these errors in our benchmark, LLM got it right (e.g., a match that is just converting enum variant name to a string), but when the error occurred in the context of more involved match, we found that LLMs came up with a fix different from the developer.

E. RQ4: Can RustAssistant *generalize to fix errors reported by a Rust static analyzer?*

Clippy comes with an auto-fix option that is based on pattern-matching and we use it as a baseline for comparison against RustAssistant. Table VI present the results, where RustAssistant is able to fix 2.4x more errors than Clippy auto-fix, achieving a peak accuracy of nearly 75%.

These accuracy numbers, however, are based on whether the generated fix makes the clippy error go away. We manually conduct further assessment to check if the fix preserves runtime behavior of the program. Clippy checks are mostly simple, and the generated fixes are usually small, making this exercise feasible. We find that the RustAssistant fix in 229/259 (88%) cases preserves the runtime behavior of the program, to the best of our judgment. In the remaining cases, we were uncertain if that is the case or not.

We illustrate some of the fixes through examples. Following is a pedantic lint error about a function argument passed as value not being consumed inside the function body:

```
error: this argument is passed by value, but not consumed in the
function body --> src/lit.rs:832:32
    |
832 |     fn parse_negative_lit(neg: Punct, cursor: Cursor) ->
      Option<(Lit, Cursor)> {
    |                              ^^^^^ help: consider taking a
      reference instead: `&Punct`
```

Clippy autofix is unable to fix the error. RustAssistant, on the other hand, changes the type of the neg parameter to &Punct and also modifies the callers of parse_negative_lit accordingly. In general, we find that clippy autofix refuses non-local fixes.

For the following error, where a cast is performed on the same type, both Clippy autofix and RustAssistant are able to fix it as suggested.

```
error: casting to the same type is unnecessary (`usize` -> `usize`)
     |
5003 |          let next = cmsg as usize + __DARWIN_ALIGN32(
      cmsg_len as usize);
     |     ^^^^^^^^^^^^^^^^^ help: try: `cmsg_len`
```

In cases where more intricate code transformations are required, particularly involving memory access or logical adjustments, Clippy's autofix option falls short. For example in the below code, Clippy complains of using offset with a usize casted to an isize, RustAssistant successfully replaces the offset method with add for pointer arithmetic.

```
error: use of `offset` with a `usize` casted to an `isize`
     |
5014 | /          (cmsg as *mut ::c_uchar)
5015 | |               .offset(__DARWIN_ALIGN32(::mem::size_of::<::
      cmsghdr>()) as isize)
     | |_________________^ help: try: `(cmsg as mut ::c_uchar).
      add(__DARWIN_ALIGN32(::mem::size_of::<::cmsghdr>()))`
```

We conclude that RustAssistant's capabilities generalize to fixing clippy warnings as well. RustAssistant performs better than clippy autofix, which is not as effective when the lint error involves code restructuring.

Summary: From our evaluation, we conclude that the pre-trained LLMs seem to have internalized the knowledge of Rust syntax and commonly used Rust idioms. They also follow the errors and come up with the relevant and intended fixes in most cases. They do, however, require careful prompt construction, and the iteration with a compiler was necessary especially for propagating changes across different parts of the code.

VI. Threats to Validity

a) Internal Threats: One internal threat is implicit bias in the manual steps that were taken by the authors for evaluating RustAssistant. We took the following steps to mitigate this bias. First, the dataset curation was done before RustAssistant was built; the implementation of RustAssistant was not changed in response to its performance on the dataset. Second, the qualitative examination of the fixes generated by RustAssistant to assess their semantic correctness (RQ3, Section V) was done via a structured consensus-based manual evaluation involving multiple evaluators, ensuring more reliable and consistent assessments.

Another potential internal threat is data contamination, where it might be possible that fixes to the compilation issues that we mined from open source might have already been included in the training data of the LLMs that we used. There is no ideal way to completely remove contamination without sacrificing real-world scenarios, given the scope of training data that is consumed for these models today. However, the fixes, especially for the top-100 benchmarks were never presented online in the form of a fix or alongside the corresponding compiler error, to the best of our knowledge. Only the fixed version of the code might appear in a later version of the repository. Furthermore, we also present vanilla LLM performance and demonstrate the delta improvement that RUSTASSISTANT provides when using the same LLM. Table IV, for instance, shows an increase from 139 (51%) to 252 (93%) solved micro-benchmarks.

b) External Threats: While the our evaluation of RUSTASSISTANT encompasses Rust errors from multiple sources, we acknowledge that the generalizability of our findings to different datasets may vary. Furthermore, the nature of API access from OpenAI implies that the LLM performance can vary over time as the models may get updated without any prior intimation, which can impact RUSTASSISTANT's fix accuracy. LLMs are also non-deterministic, thus impacting the repeatability of our results, even when the model remains the same. We partially mitigate the non-determinism by sampling multiple completions ($N = 5$) from the model to make the experiments statistically more robust.

VII. RELATED WORK

Our work can be considered as an instance of the general problem of Automated Program Repair (APR) [32], [33] that is concerned with fixing "buggy" code, given some correctness specification. We only focus on learning-based APR techniques in this section.

Traditional learning-based approaches [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44] have required supervised training data (pairs of buggy and patched code) to train custom models. This can be time-consuming and expensive. RUSTASSISTANT, on the other hand, uses pre-trained foundational models, relying on their ability to follow instructions [45]; thus, we skip the data collection requirement altogether.

The potential of LLMs as powerful APR agents has been acknowledged in previous studies [4], [5], [6], [7], [8], [9], [10], [46]. For instance, Xia et al. [46], [9] have evaluated multiple LLMs (GPT-3 series, CodeT5, etc.) to show superior performance than prior APR techniques, with larger models achieving better performance. LLMs have been used to fix incorrect solutions in LeetCode contests [10], [47] as well as fix buggy algorithm implementations [8]. These techniques require the presence of test cases, which we do not. This allows RUSTASSISTANT to be used in any scenario where the developer is ready to build their code.

In terms of fixing statically-detected errors, RING [4] considers retrieval-augmented few-shot prompting to fix syntactic errors in multiple languages. InferFix [5] uses an LLM (fine-tuned Codex) to fix errors reported by a static analysis tool (CodeQL). Pearce et al. [6] explore repairing of cybersecurity bugs. Our work differs from these in multiple dimensions. First, they rely exclusive on the model to produce the patch, whereas we use a pipeline that iterates with the compiler to arrive at the fix. Second, our focus on Rust is unique. There is relatively much lesser code in Rust compared to Java and Python that the above work had used. It is not immediately evident if LLM-based techniques will carry over to Rust without impacting their accuracy, justifying the need to study Rust errors. Third, we focus on compiler errors, allowing us to leverage the specific nature of compiler-generated error messages for crafting the prompt and improving accuracy. Our work also does not rely on fine-tuning. We do, however, believe that RUSTASSISTANT can benefit from few-shot prompting, but leave this for future work.

Recent work on a tool called RustGen [48], done concurrently with our work, consider generation of Rust code from a natural language description, and then making sure that the resulting code compiles correctly. Their solution also utilizes an iterative fixing loop with the Rust compiler. However, their problem domain is one of code generation, not fixing compiler errors in existing (large) codebases. Our prompt construction, especially with the changelog format, is novel. RustGen requires construction of Rust ASTs, which we do not. Furthermore, RustGen evaluation is restricted to only ten fixed programming tasks. Our evaluation is significantly more complete with respect to Rust compilation errors, both in real-world code as well as developer-posed questions on Stack Overflow.

GitHub Copilot: GitHub recently announced an extension to its Copilot [1] for fixing code errors. This feature can, in particular, can also be applied to compilation errors from within the IDE itself (such as vscode). This IDE integration offers an efficient and convenient way for users to invoke the tool, compared to RUSTASSISTANT that requires invoking a separate build on the command line. However, Copilot does not attempt iterative fixes (i.e., fixing subsequent errors that arise from one patch), neither does it apply a search strategy to potential multiple completions. These choices are left to the user, making it difficult to conduct a direct comparison with RUSTASSISTANT.

VIII. CONCLUSIONS

This paper presents RUSTASSISTANT as a tool for automatically generating patches for compilation errors in Rust. It demonstrates that the latest advancements in LLMs, in combination with symbolic tools such as a compiler, leads to a very effective solution for fixing code errors.

LLMs are sensitive to the prompts that they are supplied. We demonstrate the features that were needed to help the model communicate code changes and bring accuracy up from a mere 10% to nearly 74%. This evidence should add encouragement to the wave of building LLM-powered tools for software engineering.

References

[1] GitHub, "Github copilot," https://github.com/features/copilot, 2022.

[2] ——, "Github copilot-x," https://github.com/features/preview/copilot-x, 2023.

[3] Emery Berger, "Chatdbg," https://github.com/plasma-umass/ChatDBG, 2023.

[4] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, "Repair is nearly generation: Multilingual program repair with llms," CoRR, vol. abs/2208.11640, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.11640

[5] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," CoRR, vol. abs/2303.07263, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.07263

[6] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. IEEE, 2023, pp. 2339–2356. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179420

[7] ——, "Can openai codex and other large language models help us fix security bugs?" CoRR, vol. abs/2112.02125, 2021. [Online]. Available: https://arxiv.org/abs/2112.02125

[8] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in 3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022. IEEE, 2022, pp. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[9] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2023, pp. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[10] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2023, pp. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[11] OpenAI, "GPT-3.5," https://platform.openai.com/docs/models/gpt-3-5, 2023.

[12] ——, "GPT-4 technical report," https://doi.org/10.48550/arXiv.2303.08774, 2023.

[13] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, and et al., "Llama: Open and efficient foundation language models," CoRR, vol. abs/2302.13971, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.13971

[14] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, and et al., "Llama 2: Open foundation and fine-tuned chat models," CoRR, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[15] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, and et al., "Palm 2 technical report," CoRR, vol. abs/2305.10403, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.10403

[16] A. Saparov and H. He, "Language models are greedy reasoners: A systematic formal analysis of chain-of-thought," 2023.

[17] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2023, pp. 2450–2462. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00205

[18] Rust Team, "Rust," https://www.rust-lang.org/, 2023.

[19] Linux kernel development community, "Rust in linux kernel," https://docs.kernel.org/next/rust/index.html, 2020.

[20] Stack Overflow, "Stack overflow survey," https://insights.stackoverflow.com/survey/2021, 2021.

[21] Mark Russinovich, "Rust in the windows kernel," https://twitter.com/markrussinovich/status/1656416376125538304?lang=en, 2023.

[22] AWS, "Sustainability with rust," https://aws.amazon.com/blogs/opensource/sustainability-with-rust/, 2022.

[23] MSRC Team, "A proactive approach to more secure code," https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2023.

[24] Rust Analyzer Team, "Rust analyzer," https://github.com/rust-lang/rust-analyzer, 2020.

[25] Rust Team, "Rust survey," https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html, 2023.

[26] jeromefroe, "Question about a Rust compilation error on Stack Overflow," https://stackoverflow.com/questions/40299671, 2016.

[27] Rust Team, "Rust error codes index," https://doc.rust-lang.org/error_codes/error-index.html, 2023.

[28] ——, "Rust clippy static analysis," https://doc.rust-lang.org/clippy, 2023.

[29] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in Advances in Neural Information Processing Systems, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837.

[30] Rust Team, "The rust community's crate registry," https://crates.io, 2023.

[31] rust-lang, "rustfix: Automatically apply the suggestions made by rustc," https://github.com/rust-lang/rustfix, 2023.

[32] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," Commun. ACM, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[33] M. Monperrus, "Automatic software repair: A bibliography," ACM Comput. Surv., vol. 51, no. 1, pp. 17:1–17:24, 2018. [Online]. Available: https://doi.org/10.1145/3105906

[34] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 87–98. [Online]. Available: https://doi.org/10.1145/2970276.2970326

[35] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[36] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: context-based code transformation learning for automated program repair," in ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[37] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[38] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, 2021, pp. 1161–1173. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00107

[39] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, 2022, pp. 1506–1518. [Online]. Available: https://doi.org/10.1145/3510003.3510222

[40] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," ACM Trans. Softw. Eng. Methodol., vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: https://doi.org/10.1145/3340544

[41] M. Sintaha, N. Nashid, and A. Mesbah, "Katana: Dual slicing based context for learning bug fixes," ACM Trans. Softw. Eng. Methodol., vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3579640

[42] A. Connor, A. Harris, N. Cooper, and D. Poshyvanyk, "Can we automatically fix bugs by learning edit operations?" in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 782–792.

[43] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," 2023.

[44] T. Ahmed, N. R. Ledesma, and P. Devanbu, "Synshine: Improved fixing of syntax errors," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2169–2181, 2023.

[45] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, and et al., "Training language models to follow instructions with human feedback," in NeurIPS, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[46] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," 2022.

[47] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 1469–1481.

[48] X. Wu, N. Cheriere, C. Zhang, and D. Narayanan, "Rustgen: An augmentation approach for generating compilable rust code with large language models," in ICML Workshop on Deployment Challenges for Generative AI, 2023. [Online]. Available: https://openreview.net/forum?id=y9A0vJ5vuM#all