Leveraging Large Language Model to Assist Detecting Rust Code Comment Inconsistency

Yichi Zhang
zhangyichi@stu.pku.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
HCST, School of Computer Science, Peking University
China

Zixi Liu
zxliu@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Yang Feng*
fengyang@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Baowen Xu
bwxu@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

ABSTRACT

Rust is renowned for its robust memory safety capabilities, yet its distinctive memory management model poses substantial challenges in both writing and understanding programs. Within Rust source code, comments are employed to clearly delineate conditions that might cause panic behavior, thereby warning developers about potential hazards associated with specific operations. Therefore, comments are particularly crucial for documenting Rust's program logic and design. Nevertheless, as modern software frequently undergoes updates and modifications, maintaining the accuracy and relevance of these comments becomes a labor-intensive endeavor.

In this paper, inspired by the remarkable capabilities of Large Language Models (LLMs) in understanding software programs, we propose a code-comment inconsistency detection tool, namely RustC^4, that combines program analysis and LLM-driven techniques to identify inconsistencies in code comments. RustC^4 leverages LLMs' ability to interpret natural language descriptions within code comments, facilitating the extraction of design constraints. Program analysis techniques are then employed to accurately verify the implementation of these constraints. To evaluate the effectiveness of RustC^4, we construct a dataset from 12 large-scale real-world Rust projects. The experiment results demonstrate that RustC^4 is effective in detecting 176 real inconsistent cases and 23 of them have been confirmed and fixed by developers by the time this paper was submitted.

*Yang Feng is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695010

KEYWORDS

code comment inconsistency, program analysis, large language model, bug detection

ACM Reference Format:
Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. 2024. Leveraging Large Language Model to Assist Detecting Rust Code Comment Inconsistency. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3691620.3695010

1 INTRODUCTION

With global reliance on software growing every day, our software community presents desperate needs for a programming paradigm that can provide strong security and safety. The Office of the National Cyber Director at the White House has underscored the importance of adopting programming languages that minimize memory safety vulnerabilities and nominated Rust as a pertinent example [3]. Under this circumstance, Rust, as a modern systems programming language, has gained particular wide attention due to its unique mechanism for supporting software safety, performance, and concurrency features. It has been the most loved language for the seventh consecutive year in the 2022 StackOverflow survey [1].

Rust is renowned for its robust memory safety features; however, its unique memory management model introduces significant challenges in program writing and comprehension [4, 7, 14]. While Rust adopts stringent borrowing and ownership rules which mandate each piece of memory can have only one owner and references to data must adhere to strict lifetime constraints, it is capable of preventing common memory bugs such as *use-after-free* and *buffer overflows*. These rules provide guarantees for memory safety; however, they significantly steepens the learning curve for developers and thus naturally brings challenges in the program comprehension and maintenance [10, 11, 15]. Thus, to ease the program comprehension and maintenance tasks, Rust community adopts a lightweight documentation practice. It ships Rust compiler with `Rustdoc` [2] that can easily translate the code comments into documents. In this way, it significantly reduces the required documentation efforts, and ease users to understand the design and usage of Rust programs.

However, this practice inherently depends on the quality of code comments, and thus makes the interplay between code and comments pivotal, particularly in the context of enhancing comprehensibility and safety. Comments within the Rust source code can explicitly describe conditions that may lead to a panic, thus alerting developers to potential risks inherent in certain operations. This practice not only aids in code comprehension but also facilitates the transformation of inline comments into user-friendly API documentation. Through tools such as Rustdoc, these comments are systematically converted into well-structured API documentation that is both accessible and informative. Consequently, this dual function of comments serves a crucial role: it not only assists programmers in understanding the internal workings and safety implications of their code but also forms comprehensive API specifications. These specifications are invaluable for developers who rely on these APIs, as they provide clear guidelines on how to correctly and safely implement the functions, thereby reducing the likelihood of runtime errors and ensuring smoother integration.

Yet, it is hard to ensure that developers writing standardized comments as constraints or completing the code logic strictly abide by requirements of the API. Meanwhile, manually ensuring such consistency remains a timeconsuming and challenging task, especially in large-scale projects where codes evolve dynamically over time [17, 19, 30]. Inconsistencies between code and comments not only impede understanding but can also lead to misconceptions, errors, and inefficiencies in usage and development. And both code and comments are inherently unstructured data forms, which complicates the process of aligning textual descriptions with the corresponding operational logic. To address these challenges, plenty of automated techniques have been proposed in the past decade [17, 19, 21, 28, 30]. However, these techniques often generally suffer from the casual document writing style, such as acronym, jargon, poor punctuation, grammar errors, typos and so on. Moreover, existing approaches do not suite for the *panic* bugs and Rust-specific features.

To overcome the aforementioned challenges, in this paper, we propose an automated consistency detection technique, namely RustC^4, for code and comment to provide quality assurance for Rust programs. Firstly, for the comment in natural language format, we apply a large language model (LLM) [5, 26] to parse and analyze the constraints, including type checking, comparison relations, boundary and empty value determination. Then, for the corresponding code snippet, we mine code patterns specific to Rust, such as matching patterns related to panic, and then perform program analysis based on abstract syntax trees (AST) to extract the data structure of variables and the constraint relationships between variables. Finally, RustC^4 checks whether it matches with the extracted constraints from the comments. If there exists a mismatch, it implies that there may be an inconsistency between the code and comments.

To assess the effectiveness of RustC^4, we performed comprehensive experiments on 12 large-scale and open-sourced Rust projects. The experimental results show that RustC^4 achieves a detection accuracy of 176/179 (98.3%) for inconsistent code-comment pairs. In addition, for the cases where inconsistencies were detected by RustC^4, we have submitted issues and 23 of them have been confirmed and fixed by the developers until we submit this paper. Further, we analyze and classify the errors in code and comment inconsistencies, calculate the percentage of each error type in code and comments, respectively, and discuss programming paradigms for avoiding expected panic bugs. Finally, we further illustrate the effectiveness of the RustC^4 and its practicality in real software development by showing multiple detected real-world examples.

The contributions of this paper can be summarized as follows:

- **Approach.** We propose a novel approach based on LLM and program analysis to analyze and extract constraints from natural and programming languages for code and comment inconsistency detection.
- **Tool.** We implemented a detection tool called RustC^4, which has been applied in real large-scale and open-source Rust projects with a high accuracy, and some of the detected errors have already been confirmed and fixed by the developers.
- **Study.** We constructed a dataset for code comment inconsistency detection, containing 176 inconsistent and 267 consistent pairs, covering multiple error types. RustC^4's validation results on this dataset demonstrate the effectiveness of our approach.

2 BACKGROUND AND MOTIVATION

Comments are a very important part of Rust. Using Rust's official documentation tool, Rustdoc, comments can be automatically generated into comprehensive documentation that stays in sync with the code. Such documentation is not only convenient for developers to reference but can also communicate code usage guidelines and interface specifications to other developers, thereby enhancing code reusability and maintainability. Additionally, Rust comments often include test code and example code, demonstrating the correct usage and expected output of the code, which helps other developers better understand the behavior and functionality of the code.

Code comments [6, 13, 16] are not always structured in a standard, concise manner. Many comments contain numerous statements and complex language patterns, as well as jargon, non-standard expressions, and potentially grammatical errors and typos. These issues pose significant challenges for traditional NLP extraction methods. However, the advent of large language models offers a new possibility. For example, for the following comments, complex language patterns makes it extremely hard to extract comments.

```
/// # Panics
/// Panics if the sci-exponent is less than [`MIN_EXPONENT`]
/// (super::basic::floats::PrimitiveFloat::MIN_EXPONENT)
/// or greater than [`MAX_EXPONENT`]
/// (super::basic::floats::PrimitiveFloat::MAX_EXPONENT),
/// or if the precision is zero
/// or too large for the given sci-exponent
/// (this can be checked using [`max_precision_for_sci_exponent`
/// (super::basic::floats::
/// PrimitiveFloat::max_precision_for_sci__exponent)).
pub fn exhaustive_primitive_floats_with_sci_exponent_and
_precision<T: PrimitiveFloat>(sci_exponent: i64, precision: u64,)
-> ConstantPrecisionPrimitiveFloats<T> {
    assert!(sci_exponent >= T::MIN_EXPONENT);
    assert!(sci_exponent <= T::MAX_EXPONENT);
    assert_ne!(precision, 0);
    let max_precision =
    T::max_precision_for_sci_exponent(sci_exponent);
    assert!(precision <= max_precision);
```

Due to the multiple parentheses, path names, and multiple variable names contained inside the comment, it is difficult to accurately

capture all the constraint relationships using traditional natural language processing models [9, 27]. For the following comments, it is not directly expressed as "Panics if `min > max`", but rather described as "Panics if `!(min <= max)`", using the computer jargon '!' denotes the negation of a condition.

```
/// Panics if `!(min <= max)`.
pub fn clamp(&self, min: A, max: A) -> Array<A, D>
{
    assert!(min <= max, "min must be less than or equal to max");
    ...
}
```

To solve the analysis of such a complex comment and extract all the variable constraint relationships, we introduce large language model (LLM) for parsing the constraints. According to a report by UBS, till the end of January 2023, ChatGPT has over 100 million monthly active users after it has been online for only two months [25]. The GPT model integrates multiple technologies such as deep learning, unsupervised learning, instruction fine-tuning, multi-task learning, contextual learning and reinforcement learning, and uses hundreds of billions of words of English corpus on the Internet for training, achieving very good conversation results. ChatGPT is built on the initial GPT (Generative Pre-trained Transformer) model, and has been continuously upgraded from GPT-1 to GPT-4. With suitable prompt, LLM could more effectively handle challenges such as non-standard expressions, jargon, and typos compared to former NLP techniques.

For the first example in this section, LLM is able to extract 3 constraints: `sci-exponent >= MIN_EXPONENT`, `sci-exponent <= MAX_EXPONENT` and `precision != 0` through complex sentences. For the second example, the constraint is successfully extracted as `min <= max` and correctly checked in the code.

3 CONSTRAINTS DEFINITION

Constraints in comments refer to the restrictions or requirements imposed by developers on function implementation [8, 22]. These constraints are intended to guide users and developers in following certain rules or conventions when using, writing, modifying, or maintaining the code [12, 18]. The inconsistency between comments and code studied in this paper refers to the issue where the constraints on API parameters specified in comments are either omitted or incorrectly checked in the code. By studying the comments of numerous APIs and analyzing the constraints within them, we selected the three most common constraints and established a comprehensive data type, `Constraint`, for RustC^4, broadly categorizing them into three types: interval constraints of parameters, existence constraints of container elements, and boundary constraints of collections.

3.1 Interval Constraints of Parameters

This type of constraints is typically used to specify that the value of function parameters should lay in an interval. Interval constraints help other developers understand the expected behavior of the function more clearly. A common form of expression is the comment *"panic if size is 0"*, which implicitly contains an interval constraint `size != 0`. An interval constraint is often presented as a binary operation: two operands and one operator. At least one of the operands is a function parameter variable, and the other can be either a variable or a number. We formalize interval constraints that RustC^4 extracted from comments as follows:

```
<constraint> ::= <param> <op> <param>
               | <constraint> <and> <constraint>
               | <constraint> <or> <constraint>
        <op> ::= > | < | >= | <= | != | ==
     <param> ::= <param_name> | <name_constant>
               | <collection> | <collection>[<index>]
               | <bool>
<collection> ::= [<param_collection>]
<param_collection> ::= <param> | <param>, <param_collection>
     <index> ::= <integer_constant>
      <bool> ::= True | False
```

3.2 Element Existence Constraints of Container

For API functions with container type input, such as `Vector`, `Hashmap`, etc., we capture the element existence constraints of container in the code comments. This type of constraint specifies that there must be some elements in the container when passing it into the API, typically indicating that the parameter must contain valid data and cannot be empty. This ensures that the function does not exhibit unexpected behavior or errors at runtime due to encountering empty values. Common corresponding comment expressions are *"Panics iff* `projection` *is empty"* or *"panics if* `keys_builder` *or* `values_builder` *is not empty"*.

In the following example, if path is empty, the program may cause panic. However, when users receive the panic, they may be confused by what cause the panic, even if they find the corresponding line of code according to error message of IDE, they might think the nodes cause panic. It may be difficult for people who do not understand the code to infer the root cause of the panic is empty path. Therefore, we should add `assert!(!path.is_empty(), "path cannot be empty")` to notify them;

```
/// This will panic if the path is empty
pub fn example(path: Vec<i64>) {
        let mut nodes = path.iter().rev();
    ...
        let dest = nodes.next()
    .expect("nodes shouldn't be empty");
```

Element existence constraints often have two fields: a parameter variable name and a boolean value. When the boolean value is `true`, it indicates that the constraint requires the container to contain some elements; when the boolean value is `false`, it indicates that the constraint requires the container to be empty.

3.3 Boundary Constraints of Collections

We further analyze the size of the container and the internal data type constraints to form boundary value constraint analysis information for the parameters. The most common form is checking if the index is out of bounds. Common corresponding comment expressions are *"Panics if the index is out of bounds"*. Many developers may not take this seriously, believing that implicit checks are sufficient, i.e., allowing the program to crash when an index

Table 1: Prompt templates of comment constraints extraction.

Interval constraints of parameters	Existence constraints of collections	Boundary constraints of collections
Format: <param><op><param> **Prompt**: If the sentence contains <op> or the following keywords, you are required to convert them into interval constraints of the parameters. If the keyword includes *greater/larger/bigger than*, they could be converted to >; the keyword such as *less/smaller/lower than* could be converted to <; *equal to* could be converted to ==; *greater than or equal to* could be converted to >=. **Example comment**: a is greater than b **Extracted constraints**: $a > b$	**Format**: <collection>: <bool> **Prompt**: Only when expressing whether the variable is empty or the length of a structure is 0, you could extract it as an existence constraints of collections, otherwise please don't extract any other comments as existence constraint. If there is an existence constraint of collections, you are required to output a boolean value to express whether the collections should be empty or non-empty. While true corresponds to empty, false corresponds to non-empty. **Example comment**: list is empty **Extracted constraints**: $list : true$	**Format**: <index> **Prompt**: Only when comment contains sentences like "index is out of bounds" or express the same meaning, you are required to extract it as a boundary constraint of collections, otherwise please don't extract any other comments as the boundary constraints. **Example comment**: index is out of bounds **Extracted constraints**: *index*

is out of bounds. However, we believe that detecting index out-of-bounds is a necessary convention, and we demonstrate this in the experiments in Section 6.

The Rust official documentation recommends[14] using methods like `assert!` to prevent array index out-of-bounds errors, or using built-in methods like `get` instead of directly accessing elements. This approach can handle potential errors more safely, especially in scenarios where index errors may occur due to user input or other dynamic factors. For example, if a user's input index exceeds the vector's bounds, using `get` allows the program to handle the situation gracefully by returning `None` instead of crashing due to invalid input.

4 APPROACH

As shown in Figure 1, the main process of RustC^4 consists of two parts. Firstly, we propose an LLM-based code comments analysis, which leads to the extraction of constraint information for the input and return values of the APIs. The detailed extraction approach is illustrated in Section 4.1. Secondly, we apply static program analysis of the code and check whether it matches with the extracted constraints, as detailed in Section 4.2. If there exists a mismatch, it implies that there may be an inconsistency between the code and the comments.

4.1 LLM-based comment constraints extraction

We extract constraints from the code comments through the following steps:

(1) Prompt word construction: The first part of the prompt is some guidance and instructions. It starts by stating that the output cannot be affected by previous questions, return results, or examples. In experiments, it was found that previous query results for the GPT model will affect its answers to subsequent queries. This may be because GPT always tries to understand and establish context related to previous discussions, and we prompt it not to be influenced by previous queries. Then it introduces what LLM is expected to do, which is to extract constraints from incoming statements and partition them by specified types. Finally, we remind the GPT model paying special attention to the part between double quotes symbols. According to our extensive research on Rust comments, double quotes often contain constraints or related variable names, such as *"Panics if `min` is greater than or equal to `max`."* and *"Panics if `min > max`"*. This can improve the efficiency and accuracy of extracting constraints.

The second part of the prompt is about some requirements for extracting operands. First, the prompt word indicates that only the noun part should be extracted and the adjectives should be ignored. For example, *"existing validity's len"* should be extracted as `validity.len()`, which is more conducive to literal comparison of the prompt word in subsequent analysis. It then illustrates some special cases that occur multiple times and are not easily extracted into the desired format. For example, negative should be extracted as less than 0 rather than as the operand negative.

In the third part of the prompt words, four types are introduced to classify the extracted constraints. If we set the constraint types to only the above three constraint types, GPT may forcibly classify content that does not belong to these three types and should not be extracted into these three types. It may be that some OutOfBounds constraints were extracted in the previous query, and many constraints that did not belong to these three types in subsequent queries were forcibly classified as OutOfBounds constraints. Therefore, in addition to the three designed constraint types, a new type Other is also introduced to classify constraints that do not belong to the first three categories. After returning the results, constraints of this type would be discarded. The prompts then list corresponding examples for each type, providing guidance for LLM to avoid mistakenly extracting constraints of one type into another type. Among the three types, we extract the interval type first because it has the widest coverage. The prompt words for each type are listed in Table 1.

(2) Iterative query: Iteratively sends queries to GPT to extract constraints, and records the constraints and their corresponding

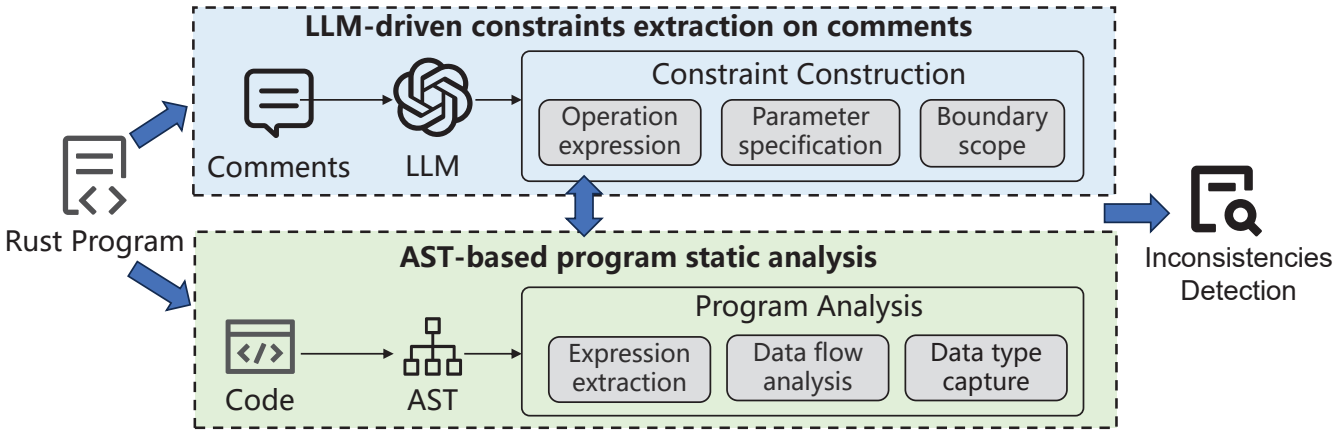

Figure 1: Overview of RustC^4.

extraction times. If the constraint is not found within the specified number of attempts, or the loop reaches its iteration limit, the loop terminates.

(3) Filter constraints: Considering that the extraction results of LLM may be unstable and the extracted content may not meet expectations, we need to filter the extracted constraints. We choose the simple, yet effective, voting strategy to identify the correct constraints from all constraints extracted from LLM's responses. Given budget limitation, we make three queries for each extraction.

4.2 AST-based program analysis

During the experiment, we found that the number of API statements tested was small and the code structure was often not complex. Using large-scale program analysis methods is expensive and complex, instead we use the syn library to construct AST for lightweight program analysis, which can obtain analysis results simply, quickly, and with less overhead. Moreover, the data flow structure of these APIs is relatively simple, so it wouldn't have much impact on accuracy.

Rust reduces aliasing issues through strict ownership and borrowing rules. Each value has one owner, preventing multiple mutable references. Rust allows either multiple immutable references (&T) or one mutable reference (&mut T), but never both, avoiding data races. Lifetimes ensure references don't outlive the data they borrow, enforcing memory safety. We maintain a table to track the flow of these pointers and references throughout the code to detect any instances where multiple identifiers refer to the same memory location, which are potential aliases.

When traversing the AST structure, constraints in the API constraint list are removed if they are detected. If a constraint is found to be incorrectly detected, it is marked as unchecked. Constraints that remain unchecked after traversal are also unchecked constraints.

Common ways to check constraints in AST are as follows:

(1) assert!: There are many macros in Rust that can be used for assertion testing. Macros are traversed through the `syn::fold::fold` macro and `syn::fold::fold_stmt` macro. RustC^4 converts the expressions checked in the assertion test into corresponding constraints according to pattern matching, and check whether they correspond to the constraints in the comments. For the first piece of code in the code snippet below, RustC^4 extracts the boolean expression from the `assert!` macro and parses it into an interval constraint. If the same constraint is found in the constraint list, it is identified as checked and removed from the list. For the latter piece of code, the constraints detected in the code are compared with the constraints in the constraint list. If the comparison symbols are found to be different, it is regarded as not detected.

```
/// # Panic
/// This function panics iff `offset + length > self.len()`.
pub fn slice(&mut self, offset: usize, length: usize) {
    assert!(offset + length <= self.len(), "the offset of the new
    array cannot exceed the arrays' length");
    unsafe { self.slice_unchecked(offset, length) };
}

/// # Panics
/// panics iff `offset + length >= self.len()`
pub fn slice(&mut self, offset: usize, length: usize) {
    assert!(offset + length <= self.len(), "the offset of the new
    Buffer cannot exceed the existing length");
    unsafe { self.slice_unchecked(offset, length) }
}
```

(2) if ... panic!(): When `panic!` occurs, set a state for the visitor. If the visitor is in a panic state when accessing the if node, the conditional expressions are obtained through the cond node of the expr_if node and converted into corresponding constraints, checking whether they correspond to the constraints in the comments. In the following example, when the expr_if node is traversed, traversing downward nodes at first. If `panic!` is traversed, the function status is set to panic and then returns to expr_if node. The panic state is recognized by fold_expr_if method and extracts the conditional expression through the AST structure and converts it into a constraint size != 0, and the constraint `size != 0` exists in the comment *"panic if size is 0"* , so the constraint is marked as checked.

```
/// Will panic if size is 0
#[must_use]
pub fn with_size(size: usize) -> SizedCache<K, V> {
    if size == 0 {
        panic!("`size` of `SizedCache` must be greater than
        zero.");
    }
    ...
}
```

(3) match ... panic: In Rust, match is a pattern matching syntax that allows programmers to execute different code branches based on different pattern matches. Patterns can consist of literals, variables, wildcards, and other things. It is common to check whether the code should panic in the arm of match pattern. In the following example, we visit the match node using the `fold_expr_match` method. First, we extract the match expression which is (self, other). Second, we traverse each arm of the match structure and extract each pattern. In the first arm, we extract the pattern (_, &Natural::ZERO) and we found panic! in the arm body and we match `other` and `&Natural::ZERO` and get a checked constraint. Third, we compare the detection constraints extracted from the match structure with the constraint extracted from the comment to detect inconsistency.

```
/// # Panics
/// Panics if `other` is zero.
fn div_mod(self, other: &'b Natural) -> (Natural, Natural) {
        if self == other {
            return (Natural::ONE, Natural::ZERO);
        }
        match (self, other) {
            (_, &Natural::ZERO) => panic!("division by zero"),
            (n, &Natural::ONE) => (n.clone(), Natural::ZERO),
        ...
}
```

(4) Call built-in API: The get method is used to safely access elements in data structures such as slices, containers, or arrays. It returns an Option type: If the index is within the valid range, it returns Some(&T), where T is the element type. If the index is out of range, None is returned. This improves the security and robustness of the code, which is a safer way to avoid the problems mentioned in the comments [14].

In addition, out-of-bounds constraints in comments are often related to operations such as inserting and deleting data structures such as containers. Built-in methods such as insert and remove will detect index out-of-bounds. Therefore, when such methods are called, it is also regarded as detecting out-of-bounds constraints.

(5) Call other methods: When calling other methods, it is very difficult and time-consuming to analyze methods located in other files. So if the variables are passed as parameters to methods in other files, the related constraints are considered undecidable and would be removed from the constraint list. To determine which method is being called within the same file and avoid calling the wrong method due to name conflicts, type analysis is used. This allows us to identify the correct method based on its type. Then, we analyze whether the constraints within the caller method have been tested in the called method, and return the results to the caller.

In the example below, the comment in `with_values` implicitly contains the constraint `values.len() == self.len()`. In the implementation, RustC^4 analyzes the `self.set_values(values)` with the parameter `values`. The caller of this method is `self`, which is of the same type as `with_values`, namely `PrimitiveArray`. We then locate the target API in the API list with the type `PrimitiveArray` and the name `set_values`. We replace the parameter variable related to the constraint in `with_values` with the corresponding parameter name in `set_values`, which is also `values` in this case. Subsequently, we analyze the AST structure of `set_values` and find that it checks the constraint using the `assert_eq!` statement. Therefore, we determine that this constraint has been successfully verified.

```
impl<T: NativeType> PrimitiveArray<T> {
    ...
    /// # Panics
    /// This function panics iff `values.len() != self.len()`.
    #[must_use]
    pub fn with_values(mut self, values: Buffer<T>) -> Self {
        self.set_values(values);
        self
    }

    pub fn set_values(&mut self, values: Buffer<T>) {
        assert_eq!(
            values.len(),
            self.len(),
            "values' length must be equal to this arrays' length"
        );
        self.values = values;
    }
    ...
}
```

The following cases can directly determine that the constraint is not checked:

(1) Obtain the context information of the index through `fold_expr_index`. If there is a boundary constraint for this index in the constraint list, it can be determined that this constraint is not checked.

(2) If a container variable still has an existence constraint when it is used, then the constraint is not checked. Here, usage refers to accessing its elements or calling methods as the caller. This indicates that the variable was not checked for being non-empty or empty before being used.

(3) For two interval constraints extracted from the code and comments respectively, if their two operands are the same and are variables rather than numbers, but the comparison operators are different, we consider this as a constraint check error. This is often due to the programmer's oversight, leading to the incorrect writing of the comparison operator in the comment or code.

5 EXPERIMENT DESIGN

In this section, we illustrate the experiment environment, subjects under test and research questions employed in our experiments.

We have implemented RustC^4 with Rust programming language, specifically version rustc 1.76.0-nightly. The newly released GPT-4.0 is selected as our LLM and we use the default temperature (T=1.0). We establish a direct connection using the OpenAI API key. All the experiments are constructed on a MacBook Pro running a macOS 13.2.1 operating system with M2 Pro Processor and 16 GB memory.

5.1 Research Questions

RustC^4 is designed to automatically detect Rust code comment inconsistency. To this end, we empirically explore the following research questions (RQ):

RQ1 (Detection Effectiveness): *How effective is RustC^4 for detecting code and comment inconsistency?* In this RQ, we aim to provide a global picture of the effectiveness of RustC^4 in detecting the inconsistency of code and comment. We apply RustC^4 to 12 real-world Rust projects and calculate the accuracy and false positive rate to evaluate the effectiveness of RustC^4.

***RQ2 (Error Types)**: What types of code and comment inconsistency bugs can RustC^4 detect?* To evaluate the diversity and adequacy of our designed constraint analysis technique, we perform further classification for the detected inconsistent instances. We categorize the causes of errors in comment and code separately and show the percentage of each category.

***RQ3 (Case Study)**: Can RustC^4 be applied to the real-world Rust projects and help developers identify inconsistency between code and comments?* As the first automated LLM-driven tool for detecting the inconsistency of Rust code and the corresponding comment, RustC^4 has been applied to detect the inconsistency in real-world projects. We discuss the practical usages of RustC^4 by analyzing some real detected examples.

5.2 Testing subjects

To the best of our knowledge, there is no standard dataset of Rust code and comment, thus, we crawled large-scale Rust projects from GitHub in order to build a dataset for test and validation. Our criteria for selecting projects are divided into two aspects. On the one hand, we choose Rust projects that are more active in the community and tend to have higher code quality, including more standardized comment writing and code logic complexity. On the other hand, we consider the diversity of the tested objects and try to cover software projects with different application scenarios as much as possible.

Table 2 introduces the detailed information of the subjects under test. All projects were up to date at the time of our evaluation. Overall, we selected 12 well-known official Rust libraries, covering various fields from Dataframe interface (`polars`) to open source cloud database (`databend`) to quantitative finance (`RustQuant`) For each project, we extracted all functions containing comments in a standardized format to construct the dataset. We then manually label each code snippet and comment for consistency with the help of dynamic runs and static code review tools. Each code-comment pair was independently checked by 3 authors with more than 3 years of experience in developing software projects, and when inconsistencies in the labeling results occurred, the actual labeling results were determined through discussion.

Specifically, through extensive observation and research of Rust API comments, we have identified that content following keywords like "Panic if" can be systematically extracted as constraints. Therefore, in the extraction, we focus on the locations of such keywords, enabling precise identification of constraint-related content and improving extraction efficiency. We only focus on the three types of parameter constraints mentioned above. Additionally, if constraints involve variables from method calls not within the same file, these constraints are removed from the list and not discussed.

5.3 Evaluation Metrics

We compare the consistency results detected by RustC^4 with the labeled ground-truth, and then count the number and percentage of the following cases:

(1) **True Positive (TP)**: The implementation of the logic in the code is inconsistent with the constraints in the comment, and RustC^4 correctly recognizes the inconsistency.

Table 2: The detailed information of crates under test.

Crate	Version	# Files	LOC	# Commits	Stars
arrow-rs	50.0.0	464	245.65k	5,717	2.3k
cursive	v0.1.1	176	35.84k	1,654	4.1k
databend	V1.2.420-nightly	3284	630.18k	31,322	7.3k
glam-rs	0.27.0	179	127.16k	988	1.4k
im-rs	15.1.0	39	18.64k	486	1.5k
janetrs	v0.7.0	26	21.23k	554	58
jsonprsee	v0.22.4	129	26.61k	792	584
pgrx	v0.12.0-alpha.1	353	303.45k	1,887	3.3k
polars	0.39.2	2,139	463.73k	9,909	27k
rerun	v15.0.1	1,809	281.08k	3,905	5.5k
RustQuant	v0.1.1	201	35.66k	851	901
uint	v1.11.1	77	13.97k	687	155

(2) **True Negative (TN)**: The implementation of the logic in the code is consistent with the constraints in the comment, and RustC^4 correctly recognizes the consistency.
(3) **False Positive (FP)**: The implementation of the logic in the code is consistent with the constraints in the comment, but RustC^4 incorrectly recognizes it as inconsistent.
(4) **False Negative (FN)**: The implementation of the logic in the code is inconsistent with the constraints in the comment, but RustC^4 fails to detect it.

To summarize, TP and TN are both implying that RustC^4 correctly detects consistency or inconsistency, while FP and FN indicate detection errors or omissions, respectively. Based on these metrics, we calculate the accuracy, precision and recall as follows.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

6 RESULT ANALYSIS

In this section, we present the experiment results on automatically detecting Rust code comment inconsistency then analyze the effectiveness and error types detected by RustC^4.

6.1 Answer to RQ1: Detection Effectiveness

RustC^4 correctly detected 176 inconsistencies between code and comments, the precision of inconsistency detection is 98.3%. Figure 2 shows a confirmed inconsistency in the project `pgrx`. In this function, the comment informed Panics if $attnum < 0$ while the code fails to check by using assert!(attnum > 0). We detected it and reported to developers, and they admitted the documentation is incorrect and fix it in the PR.

We classify the libraries in Table 2 into three scales according to the LoC, where those with less than $100k$ are categorized as small, those with more than $100k$ and less than $400k$ are categorized as medium, and those with more than $400k$ are categorized as large. In Table 3, we demonstrate the overall accuracy, precision, and recall of RustC^4 on different scale code-comment dataset. RustC^4

Figure 2: A reported inconsistency example.

achieves high accuracy on small and medium-sized projects, and a few false positives and misses on large-scale datasets, which is caused by overly complex code that results in less accurate parsing of constraints. From the average value, the detection accuracy of RustC^4 reaches more than 95%, which indicates that RustC^4 is effective in the inconsistency detection of code and comments.

Table 3: Overall evaluation results of RustC^4

Scale	Accuracy	Precision	Recall
Small (6)	97.96%	91.67%	100%
Medium (4)	99.67%	99.27%	100%
Large (2)	95.92%	96.67%	90.63%
Average	98.7%	98.32%	98.32%

The detailed detection results for each crate are displayed in Table 4. There are three false positives, which were caused by errors in the extraction of constraints and errors in the recognition of AST-based detection. For the extraction error, misjudgments occur because the constraint extraction module extracts constraints in the wrong format or extracts constraints that do not belong to the interval, element existence, and boundary types. This is often due to the GPT model's instability. For the constraint detection error, we speculate it may be that the data flow of some codes is too complex and the detection of constraints cannot be correctly identified. For example, the data flow of the statement `matches!(&validity, Some(bitmap) if bitmap.len() != self.len())` in the following code is too complex, so its detection of constraints is not recognized.

```
/// Panics iff `validity.len() != self.len()`.
pub fn with_validity(mut self, validity: Option<Bitmap>) -> Self {
    self.set_validity(validity);
    self
}

pub fn set_validity(&mut self, validity: Option<Bitmap>) {
    if matches!(&validity, Some(bitmap) if bitmap.len()
    != self.len()) {
        panic!("validity must be equal to the array's length")
    }
    self.null_bitmap = validity;
}
```

Table 4: The evaluation results of RustC^4 on different crates.

Crate	TP	TN	FP	FN
arrow-rs	14	25	1	0
cursive	2	2	0	0
databend	14	31	0	1
glam-rs	113	139	0	0
im-rs	2	5	1	0
janetrs	4	4	0	0
jsonprsee	1	2	0	0
pgrx	1	0	0	0
polars	15	34	1	2
rerun	8	1	0	0
RustQuant	1	10	0	0
uint	1	14	0	0
Total	**176**	**267**	**3**	**3**

6.2 Answer to RQ2: Error Types

According to the results in Table 5, we found that code errors are the main reason for the inconsistency between code and comments, among which boundary errors account for the absolute main reason. This is often due to developers forgetting to detect or ignoring out-of-bounds constraints, and not using built-in APIs to safely access indexes, which is contrary to the requirements of Rust official documentation. In the experiment on databend, although 5 problems were found that ignored the detection of whether index is out of bounds, it also included 20 correct detections of out-of-bounds constraints. This shows that developers consciously perform out-of-bounds constraint detection. Some of the problems we reported to developers about ignoring detection of out-of-bound constraints were confirmed and fixed by developers, which proves that detection of out-of-bound constraints is necessary.Among comment errors, comparator errors account for the main reason. This may be because developers are more likely to mistake comparators when writing comments than operands. We used RustC^4 to conduct code comment inconsistency experiments on 12 open source Rust projects on github, and detected 176 related problems, which fully proved the tool's effectiveness. Our tool also produced a small number of false positives and false negatives due to the complexity of the research problem. By calculating the metrics, we found that RustC^4 has good accuracy, precision, recall and F1 scores.

6.3 Answer to RQ3: Case Study

6.3.1 Comment Errors. There are two main sources of comment errors: one is operator error, and the other is operand error.

1. Operator error. Sometimes developers' negligence will lead to writing errors in the document. Symbol comparison errors are a common one. In the following code, the constraint `offset + length < self.len()` in the comment does not match the detection constraint `offset + length <= self.len()` in the code implementation. The reason is that the comment is written incorrectly and should be "panics iff offset + length > self.len()"

```
/// panics iff `offset + length >= self.len()`
pub fn slice (&mut self, offset: usize, length: usize) {
    assert!(offset + length <= self.len(), "the offset of the
    new Buffer cannot exceed the existing length");
```

Table 5: Statistics of the errors types detected by RustC^4.

Crate	Comment Errors		Code Errors			False Alarms	
	Operator	Operand	Boundary	Interval	Existence	Extraction Error	Detection Error
arrow-rs	4	1	9	0	0	0	1
cursive	0	0	2	0	0	0	0
databend	4	2	5	1	2	0	0
glam-rs	0	0	113	0	0	0	0
im-rs	0	0	2	0	0	1	0
janetrs	0	0	4	0	0	0	0
jsonrpsee	0	0	0	1	0	0	0
pgrx	1	0	0	0	0	0	0
polars	5	2	6	1	1	0	1
rerun	0	1	6	0	1	0	0
RustQuant	1	0	0	0	0	0	0
uint	1	0	0	0	0	0	0
Total	**16**	**6**	**147**	**3**	**4**	**1**	**2**

```
    unsafe { self.slice_unchecked(offset, length) }
}
```

2. Operand error. In addition to developer writing errors, operand errors are often due to lagging versions of comments. As projects update and iterate faster and faster, comments are often not updated with code changes, which leads to inconsistencies between comments and code. In addition, irregular operands may also be caused by irregular writing by developers. For example, in the following code, the comment should be *"Panics if `index` is greater than 3"*.

```
/// Panics if `index` is greater than 2.
pub fn col(&self, index: usize) -> Vec4D {
    match index {
        0 => [self.0[0], self.0[1], self.0[2], self.0[3]].into(),
        1 => [self.0[4], self.0[5], self.0[6], self.0[7]].into(),
        2 => [self.0[8], self.0[9], self.0[10],
        self.0[11]].into(),
        3 => [self.0[12], self.0[13], self.0[14],
        self.0[15]].into(),
        _ => panic!("index out of bounds"),
    }
}
```

6.3.2 Code Errors. Developers design APIs and write corresponding comments based on requirements, but they often mistest or miss constraints when writing code, resulting in inconsistencies between code and comment. The `Code Errors` column lists inconsistency errors due to coding issues.

1. Out-of-Bound Error. Bounded types are often constraints in code that forget to detect out-of-bounds access in comments.

```
/// Panics if index out of bounds
pub fn field(&mut self, idx: usize) -> &FieldRef {
    &mut self.fields[idx]
}
```

2. Out-of-Interval Error. This error indicates that the code logic is not written strictly according to the corresponding comments. For example, the comment *"panics if max is 0"* in the first paragraph contains the constraint max != 0, but it is not checked in the code implementation. In the second piece of code, the developer did not write the code correctly according to the requirements in the comments. The constraint in the comments is `min < max`, but the constraint `min <= max` is detected in the code implementation. After reporting it to the developer, he confirmed that it was a code error and modified the corresponding part of the code.

```
/// This function panics if `max` is 0.
pub fn max_buffer_capacity_per_subscription(mut self, max: usize)
-> Self
{
    self.max_buffer_capacity_per_subscription = max;
    self
}
/// Panics if `min` is greater than or equal to `max`.
pub fn normalize<T>(value: T, min: T, max: T) -> T where T:
FloatNumber,
{
    debug_assert!(min <= max, "min must be less than or equal to
    max");
}
```

3. Non-existence Error. The code errors of the existence type found so far are all caused by the code forgetting to check the existence constraints of the container in the comment, and the detection error that the constraint container size must be empty has not yet been recognized. For example, in the following code snippet, the comments indicate that the program could panic under some conditions, but the code doesn't contain branches for exception.

```
/// # Panics iff `projection` is empty
pub fn new(projection: &'a [usize], iter: I) -> Self {
    Self {
        projection: &projection[1..],
        iter,
        ...
}
```

7 DISCUSSION

This section discusses the application scenarios of our proposed approach and threats to validity.

7.1 Application Scenarios

This work establishes a novel approach that employs LLM and program analysis to detect the inconsistency of Rust code and comment pairs. We now briefly discuss the application scenarios of our work and the potential future research directions.

- **Automated program testing based on comment/API documentation.** The comments and API documentation can serve

as a basis for testing the correctness of programs. By extracting expected behaviors and constraints directly from documentation, RustC^4 can be applied to evaluate the correctness or soundness bugs of the programs, which can prevent the program from unexpected panic.

- **Automated testing and fixing of API documentation.** Inaccuracies in API documentation are difficult to detect but could lead to significant development delays and increased error rates. RustC^4 solves this problem by testing and subsequently fixing discrepancies between API documentation and the actual code, which can substantially improve developer productivity and software reliability.
- **Improving the program quality and assisting software maintenance.** High-quality software maintenance is pivotal for the long-term success of any software application. RustC^4 contributes to this goal by ensuring that the program code continuously aligns with its documentation. This alignment is crucial as it aids developers in understanding the code better, facilitates easier updates, and ensures consistent behavior as per the documented specifications.
- **Applying LLM to extract program constraints from complex code comments.** We demonstrate a new application scenario of LLM for software engineering, i.e., parsing complex natural language comment content to extract constraint relationships between program parameters. The ability of LLMs effectively allows them to capture detailed specifications even from casually written comments, making them indispensable tools for automated software testing systems.
- **Easy to apply to programs in other languages.** One of the significant advantages of utilizing language models in this context is their language-agnostic nature. These models can be trained to understand and interpret comments written in various programming languages. We believe this flexibility greatly enhances the portability of RustC^4, allowing it to be easily adapted and applied across different programming environments.

7.2 Threats to Validity

Even though the experiment results show RustC^4 is promising in detecting various code comment inconsistencies in Rust program, some threats may affect the validity of the results.

Internal Threats. First, one obvious threat is the fact that we only experiment on 12 projects, which may make the results difficult to be generalized to other projects. We mitigate this threat by choosing the experiment subjects from different domain. This choice may make the selected subjects representative and eliminate potential bias. Second, without a benchmark dataset for code comment inconsistency detection, we rely on manual checks, which may introduce bias and errors. To mitigate this, we assign multiple inspectors to each instance to reduce potential bias. Third, currently our tool handles only the several common forms of constraints.

External Threats. The primary external threat comes from the usage of LLM. Even though LLM can demonstrate exceptional performance in extracting constraints from comment, it may still make mistake and its behaviors and outputs are not stable. This feature may influence the accuracy of our tool. In this study, we relax this threat by conducting the experiment multiple times and report the final voting results. This strategy may reduce the potential bias and errors effectively.

8 RELATED WORK

The code-comment inconsistency problem has been studied for many years. Tan et al.[20] presented iComment, their approach combines natural language processing (NLP), machine learning, and program analysis techniques to detect code comment inconsistencies. iComment takes the first step towards automated analysis of comments written in natural language, by extracting implicit program rules and using these rules to automatically detect inconsistencies between comments and source code, thereby indicating errors or bad comments. It was evaluated on four large systems (Linux, Mozilla, Wine and Apache). At the time of publication, 60 comment code inconsistencies have been found.

In a follow-up work, Tan et al.[20] also presented @tcomment, an approach able to test the consistency between Javadoc comments related to null values and exceptions and the corresponding method behavior. @tcomment has been evaluated in multiple open source projects, and more than 10 inconsistencies identified by the tool have been confirmed and fixed by the developers. Zhong et al.[29] have proposed the first approach that detects errors for API documentations. The approach used NLP and code analysis to spot Java documentation errors. They conducted an extensive evaluation on Javadocs of five widely used API libraries. Zhou et al.[30] initiated API documentation and code discrepancy analysis and automated error detection and correction. Their tools aim at detecting inconsistencies related to parameter constraints and exceptions API between documentation and code. It was able to detect 1,146 defective document directives with around 80% precision.

Wen et al.[24] introduced GUMTREEDIFF to discloses the extent to which different types of code changes trigger updates to the related comments, identifying cases in which code-comment inconsistencies are more likely to be introduced. They mine the complete change history of 1,500 Java projects hosted on GitHub for a total of 3,323,198 analyzed commits. For each commit, they use GUMTREEDIFF to extract AST operations performed on the files modified in it. Rabbi et al.[17] proposes a solution to detect code comment inconsistency considering program and comment comprehension using a siamese recurrent network. In the proposed siamese network, there are 2 seperate RNN-LSTM models who are responsible for analyzing codes and comments and representing them into vectors. The two vectors is used to measure similarity between code and comments and inconsistency is predicted based on the similarity. Wang and Zhao[23] proposed APICAD to detect API abuse errors in C/C++ software. APICAD builds the context of API calls and mines specifications from it using a frequency-based approach. At the same time, specifications are obtained from documents by using lightweight keyword-based techniques and assisted by NLP. Finally, combination specifications are generated for error detection.

Different from these existing works on other programming languages, we proposed the first detection tool designed for Rust and aiming the Rust-specific panic bugs and type constraints. In addition, most existing tools are designed to extract the comments based on patterns or small-scale neural networks, but suffers from

the insufficient NLP understanding ability. However, our proposed approach applies the fast-growing large language model and combines it with the types of constraint relations we have mined to guarantee that the extracted constraint relations are more accurate.

9 CONCLUSION

As the requirements for program security, concurrency and high performance increase, Rust becomes more and more important. The importance of comments in Rust is self-evident. Well-written comments can help other developers understand the intent and implementation details of the code. However, many reasons can lead to inconsistencies between code and comments.

Our study introduces an approach employing LLM and program analysis to effectively detect and address inconsistencies between Rust code and comments automatically, which reduces programmer's work to check them manually. The implementation could be found in https://github.com/YichiZhang0613/RustC4. While there are many related studies in other languages, it is first time to focus on Rust. It is an innovation to leverage LLMs' advanced natural language capabilities, which demonstrates a significant step forward in automating documentation practices, ensuring more accurate and efficient developer collaboration.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their constructive comments, the github developers who responded to our post and everyone who provided suggestions for RustC^4. This project was partially funded by the National Natural Science Foundation of China under Grant No. 62372225 and No. 62172209.

REFERENCES

[1] [n. d.]. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages. (Accessed on 06/08/2024).

[2] [n. d.]. What is rustdoc? - The rustdoc book. https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html. (Accessed on 06/08/2024).

[3] 2023. White House urges developers to dump C and C++ | InfoWorld. https://www.infoworld.com/article/3713203/white-house-urges-developers-to-dump-c-and-c.html. (Accessed on 03/17/2024).

[4] William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503* (2022).

[5] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[6] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[7] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 246–257.

[8] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[9] Anusha Garlapati, Neeraj Malisetty, and Gayathri Narayanan. 2022. Classification of Toxicity in Comments using NLP and LSTM. In *2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Vol. 1. IEEE, 16–21.

[10] Ralf Jung. 2020. Understanding and evolving the Rust programming language. (2020).

[11] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.

[12] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*. PMLR, 5110–5121.

[13] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1073–1085.

[14] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

[15] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 234–245.

[16] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[17] Fazle Rabbi and Md Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th International Conference on Program Comprehension*. 371–375.

[18] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology (TIST)* 13, 5 (2022), 1–44.

[19] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. 2020. Towards Detecting Inconsistent Comments in Java Source Code Automatically. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 65–69. https://doi.org/10.1109/SCAM51674.2020.00012

[20] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/1294261.1294276

[21] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.

[22] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.

[23] Xiaoke Wang and Lei Zhao. 2023. Apicad: Augmenting api misuse detection through specifications from code and documents. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 245–256.

[24] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.

[25] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica* 10, 5 (2023), 1122–1136.

[26] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[27] Xiaoqing Zhang, Yu Zhou, Tingting Han, and Taolue Chen. 2020. Training deep code comment generation models via data augmentation. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 185–188.

[28] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 803–816. https://doi.org/10.1145/2509136.2509523

[29] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

[30] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.