Towards Understanding Rust Documentation at an Ecosystem Scale

Yufei Wu Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
wuyf21@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Rust, as an emerging programming language emphasizing both security and efficiency, introduced a novel documentation feature for developing runnable test cases, thereby improving code quality, security, and maintainability. However, it is still unknown whether and how Rust's documentation is used in practical Rust projects. Without such knowledge, Rust language designers might miss opportunities to further improve the language design, tool builders might build on incorrect assumptions, and Rust developers might miss opportunities to improve documentation quality thus incurring higher maintenance costs.**

To fill the gap, in this paper, we conduct, to the best of our knowledge, the *first* and most *comprehensive* empirical study of Rust documentation at an ecosystem scale. We first designed and implemented a novel software prototype dubbed RUSDOC, to automatically analyze Rust documentation. Then we applied RUSDOC to the entire dataset of 81,458 crates from the Rust crate registry, `crates.io`, to conduct a quantitative study to investigate the presence, completeness and inconsistency of documentation in the Rust ecosystem, which is complemented by a qualitative study to investigate the root causes leading to document-code inconsistencies. We obtained important findings and insights from empirical results, such as: 1) we identified 2 root causes leading to the low ratio of documentations; 2) we revealed 3 root causes for insufficient document testings; 3) we proposed 3 root causes leading to document-code inconsistency; and 4) we found developers prioritize the quantity of documentation over quality. We suggest that: 1) Rust language designers should elaborate on the documentation specification; 2) checking tool builders should provide effective tools to support developers; and 3) Rust developers should prioritize documentation quality. We believe these findings and suggestions will benefit Rust language designers, tool builders, and Rust developers, by providing better guidelines for Rust documentation.

***Index Terms*—Empirical study, Rust Documentation, Inconsistency**

I. INTRODUCTION

Rust [1] is an emerging programming language that guarantees both security and efficiency, by incorporating advanced language designs, a safe type system [2], and lifetime-based memory management. Specifically, Rust introduces *document testing*, a feature allowing developers to write Rust testing code in documentation. During testing, the Rust testing code in documents are compiled and executed, guaranteeing not only the normal functionalities of the testing code but also the consistencies between the testing code and the Rust code it documents. Due to the benefits brought by its double roles of documentation and testing, Rust's documentation is important in improving the Rust project's code quality, security, and maintainability, bringing considerable engineering advantages to Rust developers.

To better guide the use of documentation, Rust provides an official Rustdoc specification [3] as well as a community guideline [4], proposing three important principles any Rust documentation should follow:

- **H1:** The front-page of any `crate` documentation should have an introduction, an example code, and a detailed description of its core functionality.
- **H2:** Every public item in Rust crate, such as traits, structs, enums, functions, methods, macros, and type definitions, should have an explanation of their functionalities.
- **H3:** Every public item (as aforementioned in H2) should have an example code (*i.e.*, testing code), which exercises the expected functionalities.

We dub these principles the *RustDoc hypothesis*, as they are important assumptions that Rust developers should have followed.

Unfortunately, although the RustDoc hypothesis provides important guidelines for Rust developers, it is still unknown whether this hypothesis truly holds in practice. Instead, the current Rust community has assumed *optimistically* that the RustDoc hypothesis is already held (*i.e.*, all RustDoc principles have been followed). For example, `crates.io`[5], the largest repository for Rust's crates (Rust's terminology for packages), does not check the RustDoc hypothesis when new crates are uploaded and registered. As a result, such a lack of checking of RustDoc hypothesis might lead to software defects such as confusions [6], inconsistencies [7] [8], vulnerabilities, or even bugs [9] [10] [11] [12].

One may speculate that the study of documentation qualities and inconsistencies is a solved problem, as there have been a significant number of studies in this direction [13] [14][15] [16] [17]. However, two issues still troubled Rust developers: first, an ecosystem scale Rust documentation study is still lacking. Rust documentation, whose initial goal is to provide not only documentation but also unit testings and examples, is *optional* and *nonmandatory*. Therefore, Rust does not provide any official checking tool distributed with its official compiler `rustc`[18]. Worse yet, to the best of our knowledge, there are no third-party usable tools to check Rust's documentation,

due to the intrinsic difficulty of checking Rust code in documentation. Without such tools, it is difficult if not impossible to perform an ecosystem scale study in an automated manner.

Second, analyzing Rust documentation is challenging. As Rust documentation contains not only comments written in natural languages, but also arbitrary Rust code for testing, a study of Rust documentation needs to analyze the Rust testing code, with dedicated program analysis algorithms. Furthermore, as document testing code generally serves as unit testing, the program analysis algorithms should process the Rust code being documented simultaneously. However, prior studies on *comment-code* inconsistencies [13] [19] [20] [21] [22], utilizing techniques of natural language processing (NLP), focused on only comment qualities or inconsistencies between *comment* and code.

To this end, to study Rust documentation, several key questions remain unanswered: What amount of documentation exists in Rust crates? To what extent is the Rust documentation complete? What is the size of documentation that Rust developer write? What is the percentage of document-code inconsistencies in Rust ecosystem? What are the root causes leading to document-code inconsistencies? Does the quality of Rust documentation improve over time? What challenges do Rust developers face? Without such knowledge, Rust language designers might miss opportunities to further improve language design, tool builders may build on wrong assumptions, and Rust developers may miss opportunities to improve documentation quality and reduce code maintenance costs.

Our work. To fill this gap, this paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study of Rust documentation at an ecosystem scale by utilizing a combination of quantitative and qualitative approaches, in three steps. First, we designed and implemented a novel software prototype dubbed RUSDOC. RUSDOC has a crawler module to crawl all crates on Rust central registry crates.io for subsequent analysis. Next, RUSDOC utilized a quantitative approach to analyze each Rust crate in terms of its documentation presence, completeness, and inconsistency, in a fully automated manner.

Second, we created three datasets with 81,458 crates crawled from Rust central registry `crates.io`, and then applied RUSDOC on the datasets to conduct a quantitative study, which is complemented by a qualitative study to further investigate the root causes leading to documentation inconsistencies.

Finally, to investigate developer challenges, we conducted a developer survey to understand their perceptions of Rust documentation and challenges they encounter.

The empirical results give interesting findings and insights, such as: 1) we identified 2 root causes leading to the low ratio of documentations; 2) we revealed 3 reasons for insufficient document testings; 3) we proposed 3 root causes leading to document-code inconsistency; and 4) we found that Rust developers prioritize documentation quantity over quality.

Based on the above empirical results, we suggest that: 1) Rust language designers should clarify the documentation specification, and improve the official `rustdoc` tool[23] to strengthen its detection capabilities; 2) checking tool builders should develop specific tools to detect document-code inconsistencies more effectively; and 3) Rust developers should integrate documentation checking tools into their development workflow, to check documentation at an early development stage.

Our findings, empirical results, tools, and suggestions will benefit several audiences. Among others, they 1) provide suggestions to Rust language designers to clarify Rust documentation; 2) help checking tool builders to improve their tools; and 3) help Rust developers to detect document-code inconsistencies early.

Contributions. To the best of our knowledge, this work represents the *first* step toward a *comprehensive* understanding of Rust documentation at an ecosystem scale. To summarize, our work makes the following contributions:

- **Empirical study and tools.** We presented the *first* and most *comprehensive* empirical study of Rust documentation at an ecosystem scale, with a novel software prototype we created dubbed RUSDOC.
- **Findings and insights.** We presented empirical results, findings from the study, as well as implications for these results, future challenges, and research opportunities.
- **Open source.** We make our tool and empirical data publicly available in the interest of open science at `https://doi.org/10.5281/zenodo.7871376`.

Outline. The rest of this paper is organized as follows. Section II presents the background for this work. Section III presents the research questions that guided our experiment and the selection criteria for the Rust code that comprises our data set. Section IV presents the approach we used to perform the analysis. Section V presents empirical results, by answering research questions. Section VI and VII discuss implications for this work, and threats to validity, respectively. Section VIII discusses the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work, by introducing the Rust programming language (§ II-A), Rust documentation (§ II-B) and a motivating example (§ II-C)

A. Rust

Brief history. Rust [1] is an emerging and rapidly growing programming language, which was initially designed by Graydon Hoare in 2006, and was first publicly released in 2010 [24]. Rust released its first stable version 1.0 in 2015, and its latest stable version is now 1.68.0 (as of this study). After a decade of development, Rust has grown into a production quality language with increasing popularity.

Advanced Features. Rust emphasizes both efficiency and safety. First, Rust achieves efficiency by utilizing an explicit memory management system based on ownership [25] and a lifetime model, without any runtimes or garbage collectors.

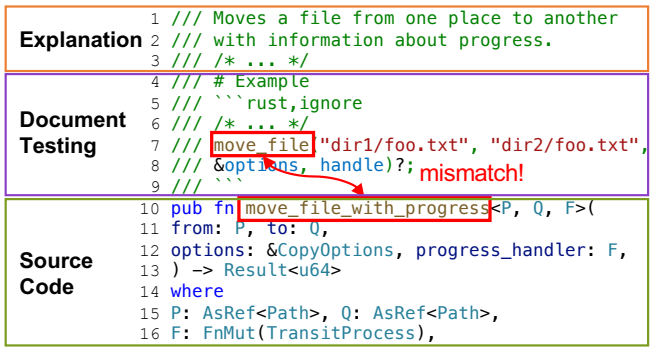


Fig. 1: A motivating example.

The ownership and lifetime are both checked and enforced statically at compile-time, eliminating potential runtime overheads. Second, Rust guarantees memory safety and thread safety through its sound type system supplemented by numerical runtime checks. Rust's type system uses linear logic [26] and aliased types [27][28] to prevent memory vulnerabilities such as dangling pointers, memory leaks, and double frees.

Wide applications. Rust, due to its efficiency and safety advantages, is widely used in diverse domains, including operating system kernels [29] [30] , Web browsers [31], file systems [32], cloud services [33], network protocol stacks [34], language runtimes [35], databases [36], and blockchains [37]. In the future, a desire to secure the cloud or edge computing infrastructures without sacrificing efficiency will make Rust a promising language.

B. Rust Documentation

Rust provides well-designed support for documentation. On the one hand, Rust harnesses successful documentation designs from other programming languages (*e.g.*, Java [38], Python [39] and JavaScript[40]), to design its own documentation specification [3].

On the other hand, Rust provides specific documentation methods and a simple-to-use tool called `rustdoc`, which generates user-friendly HTML documents. Additionally, to simplify package building and management, Rust provides the `cargo` tool [41] and eco-friendly package repositories `crates.io` [5] and `Docs.rs` [42]. Consequently, Rust's documentation is extensive and effective, reducing the learning curve for Rust programmers.

Specfically, Rust offers a novel feature called document testing, which allows programmers to write Rust code in documentation, which is automatically executed when the test is triggered. This new feature is not supported in other languages such as Java.

C. Motivating Example

To put the discussion of Rust documentation in perspective, Fig. 1 presents an illustration of a document-code inconsistency in a real-world Rust crate (`fs_extra`[43]), which is detected by our tool RUSDOC. In the figure, lines 1 to 3 are a short description of the function, which we call "explanation", which supports **H2**; lines 4 to 9 are a runnable example, which we call "document testing", which supports **H3**; lines 10 to 16 are the source code.

However, the document testing is flawed due to its incorrect invocation of a wrong function (line 7). Moreover, this flaw will not be captured by existing tools, for two reasons: 1) it uses the "`ignore`" flag, so will not be executed by the test tool `rustdoc`; 2) if the function `move_file` does exist, the tests will incorrect invoke and test the *wrong* function.

III. METHODOLOGY

In this section, we present the research questions (Section III-A) that guide our study, as well as the data selection criteria (Section III-B) for the Rust crates we chose and explored.

A. Research Questions

The main goal of our work is to conduct the first ecosystem scale empirical study of Rust documentation. To this end, we aim to answer the following high-level questions:

- Does the Rustdoc hypothesis hold in practice?
- What challenges do Rust developers face?

In the following, we refine the above questions into six research questions (RQs):

RQ1: Presence. What amount of document exists in Rust crates?

All Rustdoc hypotheses emphasize the importance of documentation. The motivation for RQ1 is to check the presence of Rust documentation, to explore whether the Rustdoc hypothesis holds.

RQ2: Completeness. To what extent is the Rust documentation complete?

The second Rustdoc hypothesis specifies that the documentation should have an explanation of their functionalities, and the third hypothesis emphasizes the importance of examples. The motivation for RQ2 is to examine the completeness of the Rust documentation, further exploring whether **H1**, **H2** and **H3** hold.

RQ3: Size. What is the size of documentation that Rust developer write?

The purpose of RQ3 is to evaluate the quality of documents in terms of documentation size. Solving this question requires a reasonable way to quantify the size of the documentation. A potential candidate is counting the number of lines per document. However, the number of document lines is an admittedly subjective notion, as mentioned in prior work[44].

Because some documents may contain meaningless characters and incomprehensible words, moreover, different blank line and newline schemes can inadvertently bias such measurements. To complement this study, we count the number of words in a document to obtain a more precise documentation size.

RQ4: Inconsistency. What is the percentage of document-code inconsistencies in Rust ecosystem? What are the root causes leading to document-code inconsistencies?

RQ4 aims to quantify and understand the extent of the inconsistency between code and documentation in the Rust

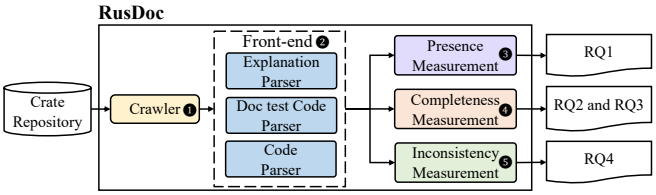


Fig. 2: RUSDOC architecture.

ecosystem. Since documentation is critical for developers to understand what code does, inconsistencies lead to confusion, errors during development and maintenance, and lost productivity. Identifying the root causes of inconsistencies is critical to developing effective strategies to prevent or mitigate them.

RQ5: Evolution. Does the quality of Rust documentation improve over time?

The motivation for RQ5 was to investigate whether the introduction of Rustdoc guidelines [4] and the increased focus on the Rust programming language in the industry had changed the practices of Rust developers.

RQ6: Rust Developer Perception. What challenges do Rust developers face?

RQ6 aims to understand Rust developers' perceptions with regard to documentation and the challenges they face when writing and consulting it.

B. Data Selection

To understand documentation in the Rust ecosystem, we analyzed real-world, publicly available Rust code. Three principles are guiding our selection criteria for Rust projects.

First, in order to cover as wide a range as possible, we included as many Rust crates as possible in our study. We selected the latest versions crates on the Rust community's crate registry (`crates.io`). Crates are the smallest unit of Rust compilation that can be compiled into libraries or binaries, depending on whether the crate includes a `main()` function.

Second, as with any open ecosystem, there is a long tail of Rust crates that are small, largely unused, and may not reflect the overall ecosystem. Therefore, we analyzed the popular and actively-maintained crates in our dataset.

Third, to compare crates contributed by the larger Rust community with crates developed by members of the Rust core development team, we analyzed the Rust standard library. The Rust standard library provides best practices on writing good documentation and document testing.

IV. APPROACH

In this section, we present our approach to conduct the empirical study. We designed and implemented a software prototype RUSDOC to mine a large-scale registry of Rust crates, to automatically generate the report of documentation quality and document-code inconsistencies. We first introduce the design goals of RUSDOC (Section IV-A), and its architecture (Section IV-B). We then discuss the design and implementation of the crawler module (Section IV-C), the front-end module (Section IV-D), the presence measurement module (Section IV-E), the completeness measurement module (Section IV-F), and the inconsistency measurement module (Section IV-G), respectively.

A. Design Goals

We have two design goals for RUSDOC on large-scale Rust datasets: 1) automation and 2) scalability. First, the study should be fully automated, otherwise it is difficult, if not impossible, to study large datasets with tens of thousands of crates in a fully automatic manner; human analysis is only required to complement the analysis by manual code inspection.

Second, the study can be applied to any Rust projects with different structures instead of specific ones. RUSDOC is designed with the principle of modularity and extensibility, so that it is straightforward to make modifications suitable for different needs, such as adding new datasets, experimenting with new research questions, or studying new evaluation metrics.

B. The Architecture

Based on the above design goals, in Fig. 2, we present the architecture of RUSDOC, which consists of five key modules. First, the crawler module (❶) crawls all crates in the crate repository, to obtain the source code and meta information of the crates, such as downloads and version numbers.

Second, the front-end module (❷) takes as input the Rust source code, and extracts three parts: the source code, the explanation, and the document testing code. Then, the front-end module parses the code into the abstract syntax tree (AST) for the next step.

Third, the presence measurement module (❸) takes as input the result of the front-end module and calculates the presence of the documentations. This result is used to answer the first research question (**RQ1**), i.e., the presence of documentation.

Fourth, the completeness measurement module (❹) takes as input the result of the front-end module and calculates the completeness of the explanations and document testings. This result is used to answer the second and third research questions (**RQ2** and **RQ3**), i.e., the completeness and size of documentation.

Finally, the inconsistency measurement module (❺) takes as input the result of the front-end module and calculates the inconsistency of document testing and code. This result is used to answer the forth research question (**RQ4**), i.e., the document-code inconsistencies.

In the following sections, we discuss the design and implementation of each module, respectively.

C. The Crawler

We designed a crawler to collect the source code of crates from the crate repository. For each crate, in addition to source code, we also collect corresponding metadata, such as crate name, version number, downloads, and release/update time. To crawl the crates.io more effectively, we utilize a method

that involves downloading the database dump index provided by the crate repository. This index provides us with the crate name and version number. Then we download the data from the crate repository, according to the crawler policy provided by it. After fetching the data, we set up a database of Rust crates that stores comprehensive metadata such as a crate's source code and version details. This database will be used for the next step.

D. Front-end

The front-end module processes Rust source files in the following three steps: 1) code filtering: removing the binary crates and leaving only the library crates, filtering source Rust code by removing components that are not irrelevant to document testing, such as tests and scripts; 2) document split: separating the code, explanations and document testings in the source code for the next step of parsing; and 3) AST generation: the document testing code parser and the code parser take as input the Rust code, and build the Rust abstract syntax trees (AST). The AST is a tree representation of the source programs, especially; in particular, it contains necessary documentation information for subsequent analysis.

Although it is possible to combine the front-end with other phases, the current design of RUSDOC, from a software engineering perspective, has two key software engineering advantages: 1) it makes RUSDOC feasible to process different Rust crates with different structures; and 2) it makes RUSDOC more efficient in detecting defects by removing irrelevant files at an early stage.

E. Presence Measurement

All of the RustDoc hypotheses are intended to illustrate the importance of documentation, but it is unknown whether documentation exists. To this end, RUSDOC incorporates a presence measurement module to calculate the presence ratio. The presence ratio is further used to answer **RQ1**.

To analyze the presence, we took a three-pronged approach. First, we identified every presence of a document in the dataset, and specifically counted the numbers of crates containing any documentation. That is, we count how many crates contain at least one documented item. We count the number of crates with documentation $\#\texttt{doc}$, the number of all crates $\#\texttt{total}$, and then calculate $\frac{\#\texttt{doc}}{\#\texttt{total}}$ to determine the number of crates with documentation in the entire dataset.

Second, evaluating presence based solely on whether a crate contains arbitrary documents can give a rough impression. As compensation, we supplement these data by measuring documentation ratio in each crate, i.e., the ratio of documented public items to all public items. We count the number of documented items $\#\texttt{item}_{\texttt{doc}}$, the number of public items $\#\texttt{item}_{\texttt{pub}}$, then calculate $\frac{\#\texttt{item}_{\texttt{doc}}}{\#\texttt{item}_{\texttt{pub}}}$. While not a perfect metric, we believe the ratio of documentation is representative of the presence of Rust documentation.

Third, we count the presence of documentation on each public feature (module, trait, struct, enum, function, method, macro, and type definition) in the dataset to understand the presence of documentation on each feature. We count the number of documented features $\#\texttt{feature}_{\texttt{doc}}$, the number of public features $\#\texttt{feature}_{\texttt{pub}}$, then we calculate $\frac{\#\texttt{feature}_{\texttt{doc}}}{\#\texttt{feature}_{\texttt{pub}}}$.

F. Completeness Measurement

The second and third hypotheses of RustDoc require comprehensive documentation to include explanations and document testings. To this end, RUSDOC incorporates a completeness measurement module to calculate the completeness ratio. This ratio measures the completeness of explanations and document testings by taking as input the AST generated by the front-end module and the corresponding source code. The completeness ratio is further used to answer **RQ2** and **RQ3**.

To count the completeness of the documentation, we separately count whether there is an explanation and a document testing on each documented item. We count the items with explanations $\#\texttt{explanation}$ the items with document testings $\#\texttt{testing}$, and the items with either or both of them $\#\texttt{item\_doc}$. Then we calculate $\frac{\#\texttt{explanation}}{\#\texttt{item\_doc}}$ and $\frac{\#\texttt{testing}}{\#\texttt{item\_doc}}$ to obtain the completeness ratio of each item.

G. Inconsistency Measurement

Document testings should be consistent with the corresponding code, otherwise code comprehension will be hindered. To this end, RUSDOC incorporates an inconsistency measurement module to calculate the inconsistency ratio. This ratio measures the inconsistency by taking as input the AST generated by the front-end module. The inconsistency ratio is further used to answer **RQ4**.

Different types of features require varying standards to assess consistency. For instance, a function should evaluate its name, input arguments, and output. If an error occurs during function calls in document testing, such as calling the wrong function name, it indicates a potential inconsistency between the document and code, while the lack of errors establishes their consistency. To facilitate subsequent empirical studies, the inconsistency measurement module has two key functionalities: 1) analyze documents with different methods according to different types of features to find potential inconsistencies; and 2) extract detailed inconsistency information (e.g., file path, line number) and generate a uniformly formatted inconsistency summary.

To calculate the inconsistency ratio, we first count the total number of all document testings $\#\texttt{total}$, then we count the number of inconsistencies $\#\texttt{mismatch}$, then calculate $\frac{\#\texttt{mismatch}}{\#\texttt{total}}$.

V. EMPIRICAL RESULTS

In this section, we present the empirical results by answering the research questions. We first illustrate the experimental setup (Section V-A), then present the datasets (Section V-B), and then answer the previously mentioned research questions (Section V-C to Section V-H)

TABLE I: Dataset used in this study.

Name	Source	Size
DS1	crates.io [5]	81,458 crates
DS2	crates.io	500 crates
DS3	Rust standard library [45]	404 files

A. Experimental Setup

All the experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 12 GB of RAM running Ubuntu 21.04.

B. Datasets

We created three datasets, as shown in Table I. Our data selection criteria are presented in Section III-B. Here we present the crates actually included in our study and describe reasons why we are unable to include all Rust crates in our dataset.

First, we selected the latest versions of 81,458 crates on the Rust community's crate registry (crates.io). For the total 109,379 crates on crates.io, we excluded 13,580 crates whose latest versions were not compiled successfully. As the aim was to analyze the quality and inconsistency of the crate documentation, we excluded crates that were compiled into binary (14,341 in total). It should be noted that Rust is under development and a particular version of crate can only be compiled in a particular version of the compiler. For crates with conditional compilation, we use the default flags specified in the manifest. Afterwards, our dataset DS1 contains 81,458 crates, which represent 74% of the total registered crates.

Second, we analyzed the popular and actively-maintained crates in our dataset. The popularity of crates was measured based on the number of downloads. For this analysis, we selected 500 crates whose downloads accounted for 80% of the total downloads, indicating popularity - a criterion used in prior work [46]. We call this dataset DS2.

Third, we selected version 1.68.0 of the Rust standard library for our study because it was the most latest version at the time of our research. We call this dataset DS3.

C. RQ1: Presence

To answer RQ1 by investigating the presence of the documentation, we first apply RUSDOC to the dataset DS1. TABLE II shows, in absolute and relative quantities, how many crates contain no document and at least one document. Overall, 61.9% of crates contained at least one document, while 38.1% of crates had no document at all.

Second, we apply RUSDOC to the datasets DS1 and DS2, to obtain the documentation ratio of crates. Fig. 3 shows the cumulative distribution of the documentation ratio per crate for datasets DS1 and DS2.

Third, we apply RUSDOC to the datasets DS1, DS2 and DS3, to obtain the documentation ratio of each feature. TABLE III shows the documentation ratio of different features, in different datasets. The first column lists each feature that

TABLE II: Rust crates with and without any document grouped by feature. A crate may contain multiple documented features.

Document	#Crates	Ratio
None	31034	38.10%
Some	50424	61.90%

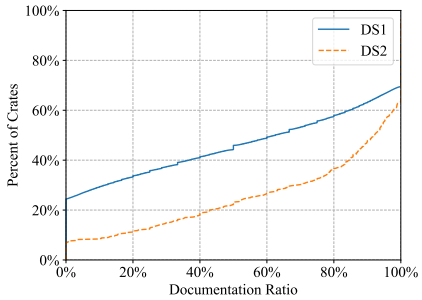


Fig. 3: Cumulative distribution of documentation ratio.

should have documentation. Note that this item must be public, since only public items will be exposed externally. The second, third and fourth columns list the percentage of items that have documentation in the DS1, DS2 and DS3, respectively.

The empirical results give two interesting findings and insights. First, the documentation ratio of all items is less than 35% in all datasets, which means that more than half of the items have no documents. However, the document ratio of DS2 (34.9%) and DS3 (34.00%) is significantly higher than that of DS1 (11.52%).

Second, traits have the highest ratio (52.49% in DS1), while type definitions have the lowest ratio (2.98% in DS1).

To further explore the potential reasons, we perform a root cause investigation by manual inspection of the source code. This inspection revealed two key reasons. First, the popularity of crates tends to be influenced by the availability of documentation. Consequently, an extensive crate documentation can cater to a broader readership. This contributes to their widespread use.

Second, trait is a Rust feature that specifically outlines how different types should behave and interact. Trait users are more likely to consult the documentation; in contrast, type definitions directly and clearly explain the specific functions and behavior of each type, which eliminates the need for further documentation. For example, the type alias for Result in `serde_json` is as follows:

```
1 pub type Result<T> = Result<T, Error>;
```

which means Result<T>is a type alias of Result<T, Error>.

TABLE III: Presence ratio of documentation for public features.

Feature	DS1	DS2	DS3
Module	16.54%	56.04%	78.16%
Function	38.58%	47.54%	25.49%
Struct	14.28%	22.38%	34.51%
Enum	18.81%	28.00%	67.74%
Trait	**52.49%**	**79.32%**	**92.55%**
Method	11.24%	69.76%	36.31%
Macro	51.47%	65.54%	80.00%
Typedef	**2.98%**	**2.30%**	**5.30%**
Average	**11.52%**	**34.90%**	**34.00%**

Summary: The documentation in the Rust crate repository is usually inadequate (below 35%), Among them, traits have the highest documentation ratio (52.49% in DS1), and type definitions have the lowest ratio (2.98% in DS1). Overall, the documentation for Rust crates is in need of improvement.

D. RQ2: Completeness

To answer RQ2 by investigating the completeness of the Rust documentation, we apply RUSDOC to the dataset DS1. The empirical results are presented in TABLE IV. The first column lists public features. The second column indicates the ratio of explanation, in all documented items, respectively. The third column indicates the ratio of document testing.

The empirical results give three interesting findings and insights. First, for public modules, the document testing ratio is 6.99%, which violates the first Rustdoc hypothesis (**H1**). Second, the explanation ratio of all items is higher than 99%, that is, most documents contain a description in natural language. This partially supports Rustdoc's second hypothesis (**H2**). Third, except for functions (14.30%) and macros (42.34%), the document testing ratio of all items is less than 10%, that is, most items do not contain document testings. This violates the third Rustdoc hypothesis (**H3**).

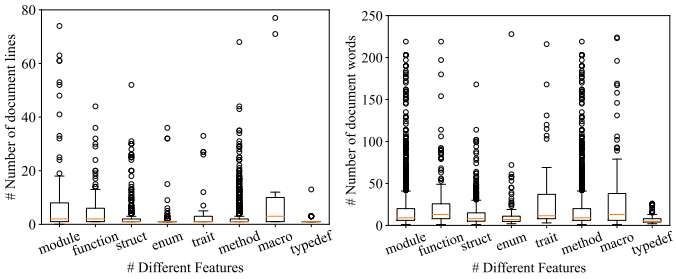
We then investigated the root causes leading to the incomplete documentation, based on manual code inspections. This inspection revealed three root causes. First, some item use-cases are simple and demand minimal elucidation, prompting programmers to disregard detailed examples while documenting them.

Second, some libraries are the underlying implementation of others, resulting in no interaction with users and a lack of examples. For instance, trust-dns-protocol [47] serves as the foundational DNS protocol library and implementation of Trust-DNS, lacking examples. Users probably do not require this library unless they intend to manipulate DNS packets directly.

Finally, we speculated that this might be due to the developer's subjective reasons that the documentation was not written. To confirm this speculation, we conducted a questionnaire survey in **RQ6** and answered this question.

TABLE IV: Explanation and document testing ratio for documented public features.

Feature	Explanation	Document Testing
Module	99.99%	6.99%
Function	99.90%	**14.30%**
Struct	99.98%	5.47%
Enum	99.99%	3.31%
Trait	99.95%	6.88%
Method	99.98%	6.07%
Macro	99.52%	**42.34%**
Typedef	99.99%	0.77%
Average	99.97%	6.43%



(a) Document lines (b) Document words

Fig. 4: The boxplot of the document lines and document words for dataset DS1.

Summary: The Rust crates has an explanation ratio of 99.97% for documented items, which partly supports the second Rustdoc hypothesis (**H2**). However, the repository has a low ratio of document testings (6.43%), which violates the third Rustdoc hypothesis (**H3**). We recommend that developers increase the writing of document testings, as this can improve the readability and usability of documentation.

E. RQ3: Size

To answer RQ3 by investigating the size of the source code documentation, we apply RUSDOC to the dataset DS1. The boxplots in Fig. 4 show the distribution of document line and word counts.

The empirical results give two interesting findings and insights. First, a majority of lines in the documents have fewer than 10 lines and have fewer than 50 words. Second, macros have the largest documentation size, while type definitions have the smallest.

We then investigate the root causes for the low-size documentation, through further automated inspection. We found a large proportion (i.e., 65.07%) of tiny documents that occupy only 1 line with a simple one-sentence description of the corresponding code.

Summary: The size of documents is typically small as the number of lines is less than 10 and the number of words is less than 50. Assuming that the line count and word count closely correspond with the level of comprehensiveness of the documentation, we speculate that most developers maintain the simplicity of their documents.

F. RQ4: Inconsistency and Root Cause Analysis

To answer RQ4 by investigating the inconsistencies of the source code documentation and analyzing root causes, we apply RUSDOC to the datasets DS1, DS2 and DS3.

The empirical results give three interesting findings and insights. First, among all crates with document testing on dataset DS1, the overall inconsistency ratio is 3.42%. Second, the dataset DS2 has an inconsistency ratio of 2.07%. Finally, we found that over 700 functions and methods in the Rust standard library with testings were examined, and the documentation for only one method did not match its source code.

To further explore the root causes leading to inconsistency factors, we performed a manual inspection of source code, and identified three key reasons. First, documentation and code may be written out of sync, leading to potential mismatches between them. This could occur when documentation is added after the code has been implemented and not thoroughly tested. For example, a document-code inconsistency occurred in crate `config_struct` [48] in the following code:

```
/// config_struct::create_struct_from_source
pub fn create_enum_from_source<S: AsRef<str>, P:
    AsRef<Path>>
```

As can be seen from the repository's commit history, all documentation was added in one commit at one time, and the author may not have noticed the mismatch.

Second, document testing is underappreciated and underutilized. Since multiple functions may have similar functionality, it is possible for a developer to copy and paste the documentation for one function into another without any modification. For instance, our tool detected a document-code inconsistency in the Rust standard library, and the code is as follows:

```
// The function is not called in the associated
    document testing.
// std/sync/condvar.rs: line 66
pub fn timed_out(&self) -> bool {
```

The document testing for this function does not mention the `timed_out` method, resulting in a mismatch between the documentation and code. Since the Rust standard library provides many basic functions and common data structures, it is widely used in the Rust ecosystem, such an inconsistency may lead to misunderstanding of the code, which may further lead to bugs or vulnerabilities. We reported this issue [49] to the developers. Developers have fixed the bug, and it is now awaiting merging [50].

Third, the abuse of Rust's re-export mechanism can result in inconsistencies between the documentation and code. For example, a document-code inconsistency occurred in crate `primal_check` in the following code:

TABLE V: documentation presence ratio, completeness ratio, and inconsistency ratio by year.

Ratio	2018	2019	2020	2021	2022	2023
$\mathbf{P}^1$	6.48%	7.09%	7.79%	7.79%	12.77%	11.53%
$\mathbf{C}^2$	10.29%	10.12%	10.10%	10.08%	5.94%	6.21%
$\mathbf{I}^3$	2.01%	2.57%	2.78%	2.53%	2.70%	3.42%

[1], [2], and [3] are abbreviations of Presence, Completeness and Inconsistency, respectively.

```
/// ```rust
/// assert_eq!(primal::is_prime(1), false);
/// assert_eq!(primal::is_prime(2), true);
/// ```
pub fn miller_rabin(n: u64) -> bool {
```

The developer admitted that this is a "weird situation", since `is_prime` is a re-export of `miller_rabin` [51].

Summary: The inconsistency ratio between documentation and code is under 4%. As the popularity of the crate increases, the inconsistency ratio gradually decreases. Inconsistency occurs due to a combination of writing errors by the developer and delayed updates to the documentation. We reported some inconsistencies to the developers and received partial responses.

G. RQ5: Evolution

To answer RQ5 by investigating documentation evolution, we obtained data by retrieving all available crates on `crates.io` from 2018 to 2023, since 2018 is the earliest recorded year. We then analyzed the presence, completeness, and inconsistency. The empirical results are presented in TABLE V. The table comprises several terms, among which is presence, indicating the ratio of documentation for all public items in the dataset. The completeness ratio represents the complete ratio of document testings of these public items. The inconsistency ratio represents the number of document testing mismatches $\#\texttt{mismatch}$ divided by the number of all document testings $\#\texttt{total}$, i.e. $\frac{\#\texttt{mismatch}}{\#\texttt{total}}$.

The empirical results give three interesting findings and insights. First, the presence of documentation increases from 6.48% in 2018 to 11.53% in 2023, and we speculate that this increase is due to developers paying more attention to documentation, acknowledging software engineering advantages.

Second, the completeness ratio of the documentation decreased significantly in 2022 from 10.08% to 5.94%, due to the lack of document testing. To investigate the root cause, we conducted experiments on the 17,685 new crates released in 2022. The results revealed that the presence ratio among the new crates was 34.27%, while the completeness ratio of the document testing was merely 1.99%.

Third, the document-code inconsistency ratio increased from 2.01% in 2018 to 3.42% in 2023.

TABLE VI: Challenges faced by developers when consulting documentation.

Problem	Description	Ratio
Nonexistence	No documents on this item.	78.57%
Unexplained	The example was insufficiently explained.	53.57%
Incompleteness	No runnable example.	35.71%
Ambiguity	The description was very unclear.	32.14%
Redundancy	API description was too extensive.	21.43%
Obsolescence	The documentation is outdated.	7.14%
Incorrectness	Some information was incorrect.	7.14%

Summary: We concluded that there was no significant trend in documentation use over the past six-year period. The presence ratio (from 6.48% to 11.53%) and inconsistency ratio (from 2.01% to 3.42%) of documents have increased, and the complete ratio has decreased from 10.29% to 6.21%. Overall, developers tend to pay increasing attention to adding documentation; however, they often neglect to supplement it with adequate testing.

H. RQ6: Rust Developer Perception

To answer RQ6, we created a survey and posted it on the Rust Subreddit [52] and Rust user forum [53], a selection used in prior work [46], collecting data from 30 respondents (as of this study). The survey asked them about the problems they encountered while writing and consulting documentations.

The first question (**SQ1**) asked whether or not Rust developers add document testings to their crates. The majority (72%) answered that they added a few, and a minority (28%) responded that they added none or all, which fits our answers to **RQ1** and **RQ2**.

The second question (**SQ2**) asked Rust developers to choose one or more reasons why they did not add testings in the documentation. More than half of respondents (63%) reported not having sufficient time to write the testing, whereas a portion of them (53%) claimed to either add it or intend to do so in the future. 8% of the respondents expressed concerns about document testings, stating that they are executed relatively slowly or that some aspects cannot be effectively tested through examples. Additionally, they opined that utilizing untested examples is not a good practice. Other reasons selected include: a belief that the API's clarity was adequate without requiring an example (38%), and not knowing how to add it (8%).

The third question (**SQ3**) asked about the difficulties encountered in writing the testings. The majority of respondents (63%) reported a lack of understanding regarding how to write and test effective examples in Rust. Others indicated that the addition of document testings would considerably decelerate testing or that there was no support or feedback of official tools (such as clippy [54], rustfmt, and rust-analyzer) and the third-party tools. The responses for (**SQ2**) and (**SQ3**) further reveal root causes in (**RQ2**).

The fourth question (**SQ4**) asked about the problems encountered while consulting other crate documentation. TABLE VI shows the typical answers to this question. The first column shows the problem encountered, the second column is a short description of the problem, and the third column shows the percentage of respondents who encountered this problem. This question allows for multiple responses, hence each person can select more than one option.

Finally, the question **SQ5** asked for suggestions of Rust documentation. It is worth noting that this question is optional hence not everyone answered it. Here are some typical answers for this question, which may not necessarily be representative of everyone's views or experiences: 1) "I prefer many small examples than some large ones. small examples might not highlight all features, but it's better to have examples for each item than only large ones for some. Quantity over quality." 2) "A little effort goes a long way. Often a single-line description is enough to let your users know how the item is meant to interact with the rest of the crate." 3) "Do not repeat words from the function name. Explain technical terms."

Summary: Most Rust developers (63%) state that they do not have sufficient time to provide examples. The most (63%) challenging aspect that they encounter is not knowing how to get started. The notable challenges arising from consulting documentation are the lack of documentation (78.57%) and unclear explanations (53.57%). Developers prioritize the quantity of documentation over quality.

VI. Implications

This paper presents the first and comprehensive empirical study of Rust documentation at an ecosystem scale. In this section, we discuss some implications of this work, along with some important directions for future research.

For Rust language designers. The findings from this study offer insights for Rust language designers to enhance Rust documentation: 1) the documentation specifications should be clarified based on the results of this research; 2) the `rustdoc` tool should be improved by including document testing checks for functions and macros; and 3) the tool can be utilized to conduct completeness and inconsistency documentation checks before they are published to `crates.io`.

For checking tool builders. We recommend that checking tool builders build corresponding tools to detect real-world document-code inconsistencies. Since our prototyping system took the first step toward it, it was feasible to develop such a tool.

For Rust developers. Although the Rustdoc guidelines have been proposed for a long time, our findings support them only partially, and it is important to note that we found that the lack of testings in the documentation became more common. Based on the observation, we suggest that Rust developers should: 1) detect the weakness in the documentation they produce; and 2) improve their crates by providing more document testings to enable testing more effectively. Meanwhile, with research progress on automatic comment generation [55] [56] [57] [58], generating documentation through automated tools is becoming promising.

For researchers. This paper opens multiple paths for future works. We need to further analyze the inconsistency between explanation and the code to achieve further documentation quality analysis. Furthermore, the datasets created by our work can benefit subsequent works.

VII. Threats to Validity

As in any empirical study, there are threats to the validity of our work. We attempt to remove these threats where possible, and mitigate the effects when removal is not possible.

Data sets. The main threat to validity is related to the representativeness of our dataset. For all currently compiled library crates from `crates.io`, we believe our analysis is representative of the Rust ecosystem[44] [46]. However, in our study, our dataset only includes all currently compilable crates in `crates.io`, excluding other code sources such as GitHub. Fortunately, the architecture of RusDoc (Fig. 2) is neutral to any specific dataset used, so a new dataset can always be added without difficulties. On the other hand, our dataset may include crates that are no longer maintained. Although, these crates are not flawed as a source for our research, the older they are, the less they contribute to the current Rust ecosystem.

Measurement metrics. Our results mainly focused on static detection of inconsistencies, while ignoring dynamic detections. For example, there might be functionality inconsistencies between the documentation and the code. To mitigate this risk, we supplement our automated measurements with manual code inspections.

Natural language inconsistency detection. In our study, we only analyzed inconsistencies in the document testing, excluding the natural language part of the document (i.e., the explanation), which may potentially lead to an underestimation of the overall ratio of inconsistencies. As the Rust ecosystem is relatively new, with over 80,000 crates developed in less than four years, outdated documentation might not be prevalent. We leave it a future work to study document-code inconsistency, by leveraging prior work in this direction [20] [21] [22].

Errors in the implementation. Most of our results are based on the RusDoc framework. Errors in the implementation could invalidate our findings. To mitigate this risk, we subjected all implementations to careful code reviews and tested them extensively.

Sample bias in questionnaires. Our questionnaire survey may have sample bias. For instance, the respondent pool in the sample may not be representative of the Rust developer population. Additionally, the sample size may be too small, which could potentially affect the survey's accuracy. Answer bias and questionnaire design bias are also possible sources of error. To improve the survey's representativeness, we will continue to conduct surveys in the future.

VIII. Related Work

In recent years, there have been a significant number of studies on language documentation studies. However, the work in this study represents for a novel contribution to this field.

Documentation studies. Many research efforts have been devoted to analyzing the documentation. Steidel et al.[59] adopted machine learning methods to classify comments into seven categories, and further analyze and evaluate the quality of code comments. Geng et al. [60] proposed Fosterer as an automated solution to analyze which parts of the code have a semantic relationship to a particular token in a comment. Vidoni [61] mined and analyzed 379 systematically selected open source R packages for their quality in terms of the existence, distribution, and completeness of their documentation.

However, they did not conduct a large-scale empirical study of the Rust ecosystem.

Document-code inconsistency studies. Document-code inconsistency has been studied extensively. Tan et al. [13] proposed iComment, a technique using NLP, machine learning, and program analysis to detect inconsistencies related to the use of locking mechanisms in code and their description in comments. Then they [62] further proposed @TCOMMENT, an approach for testing Javadoc comments, specifically method properties about null values and related exceptions. Khamis et al. [14] proposed JavadocMiner, which aims to evaluate the language quality used in Java comments and the consistency between source code and comments through a set of simple heuristics. Stulova et al. [19] proposed a technology based on the mapping between source code and its documentation to automatically detect code-comment inconsistencies during code evolution. Ratol et al. [63] propose a rule-based approach called Fraco, to detect inconsistencies in code comments due to renamed refactoring operations performed on identifiers. Steinbeck et al. [16] analyzed Javadoc comments from 163 different open source projects, focusing on detecting different types of Javadoc violations and the source code elements affected by them. TDCleaner [64] is a novel model designed to automatically detect and remove obsolete TODO comments in software projects

However, all these studies cannot be used directly to investigate Rust document testing, due to the feature discrepancies between Rust and these languages.

IX. Conclusion

This paper presents the first and most comprehensive empirical study of Rust documentation at an ecosystem scale. By designing and implementing a software prototype RusDoc, we conduct a quantitative study to investigate the presence, completeness, and inconsistency of documentation in the Rust ecosystem, which is complemented by a qualitative study to investigate the root causes leading to document-code inconsistencies. We identified root causes leading to low ratio of documentations, insufficient document testing, and document-code inconsistency. We also surveyed Rust developers' perceptions of document testing to understand challenges. This work represents a first step towards understanding Rust documentation culture among the Rust community, thereby calling to further strengthen the Rust ecosystem and establish good documentation guidelines.

References

[1] "The Rust Programming Language," https://doc.rust-lang.org/stable/book/.

[2] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Nov. 2014.

[3] "How to write documentation - The rustdoc book," https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html.

[4] "Documentation - Rust API Guidelines," https://rust-lang.github.io/api-guidelines/documentation.html.

[5] "Crates.io: Rust Package Registry," https://crates.io/.

[6] J. Y. Khan, Md. Tawkat Islam Khondaker, G. Uddin, and A. Iqbal, "Automatic Detection of Five API Documentation Smells: Practitioners' Perspectives," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, Mar. 2021, pp. 318–329.

[7] H. Zhong and Z. Su, "Detecting API documentation errors," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. Indianapolis Indiana USA: ACM, Oct. 2013, pp. 803–816.

[8] S. Lee, R. Wu, S.-C. Cheung, and S. Kang, "Automatic Detection and Update Suggestion for Outdated API Names in Documentation," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 653–675, Apr. 2021.

[9] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, Nov. 2009.

[10] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and Software Bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, Oct. 2012.

[11] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? a study on the Android SDK," in *Proceedings of the 22nd International Conference on Program Comprehension*. Hyderabad India: ACM, Jun. 2014, pp. 83–94.

[12] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs Documentation and Code to Detect Directive Defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 27–37.

[13] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 145–158, Oct. 2007.

[14] N. Khamis, R. Witte, and J. Rilling, "Automatic Quality Assessment of Source Code Comments: The JavadocMiner," in *Natural Language Processing and Information Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6177, pp. 68–79.

[15] L. Pascarella, A. Ram, A. Nadeem, D. Bisesser, N. Knyazev, and A. Bacchelli, "Investigating type declaration mismatches in Python," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Mar. 2018, pp. 43–48.

[16] M. Steinbeck and R. Koschke, "Javadoc Violations and Their Evolution in Open-Source Software," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, Mar. 2021, pp. 249–259.

[17] W. Ouyang and B. Hua, "$\hat{\}\prime\}\mathbf{R}$: Towards Detecting and Understanding Code-Document Violations in Rust," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Wuhan, China: IEEE, Oct. 2021, pp. 189–197.

[18] "What is rustc? - The rustc book," https://doc.rust-lang.org/rustc/what-is-rustc.html.

[19] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards Detecting Inconsistent Comments in Java Source Code Automatically," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 65–69.

[20] T. Steiner and R. Zhang, "Code Comment Inconsistency Detection with BERT and Longformer," Jul. 2022.

[21] R. Li, Y. Yang, J. Liu, P. Hu, and G. Meng, "The inconsistency of documentation: A study of online C standard library documents," *Cybersecurity*, vol. 5, no. 1, p. 14, Dec. 2022.

[22] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-In-Time Obsolete Comment Detection and Update," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 1–23, Jan. 2023.

[23] "What is rustdoc? - The rustdoc book," https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html.

[24] G. Hoare, "Project Servo," http://venge.net/graydon/talks/intro-talk-2.pdf, 2010.

[25] D. J. Pearce, "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, pp. 3:1–3:73, Apr. 2021.

[26] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, Jan. 1987.

[27] J. Boyland, "Alias burying: Unique variables without destructive reads," *Software—Practice & Experience*, vol. 31, no. 6, pp. 533–553, May 2001.

[28] D. Clarke and T. Wrigstad, "External Uniqueness Is Unique Enough," in *ECOOP 2003 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, L. Cardelli, Ed. Berlin, Heidelberg: Springer, 2003, pp. 176–200.

[29] "Tock Embedded Operating System," https://www.tockos.org/.

[30] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring Rust

for Unikernel Development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. Huntsville ON Canada: ACM, Oct. 2019, pp. 8–15.
[31] "Servo," https://servo.org/.
[32] "TFS," https://github.com/redox-os/tfs.
[33] "TTstack," https://github.com/rustcc/TTstack.
[34] "Smoltcp: A smol tcp/ip stack," https://github.com/smoltcp-rs/smoltcp.
[35] "Tokio - An asynchronous Rust runtime," https://tokio.rs/.
[36] "TiKV: Distributed transactional key-value database, originally created to complement TiDB," https://github.com/tikv/tikv.
[37] "Parity-ethereum: The fast, light, and robust client for Ethereum-like networks," https://github.com/openethereum/parity-ethereum.
[38] D. Kramer, "API documentation from source code comments: A case study of Javadoc," in *Proceedings of the 17th Annual International Conference on Computer Documentation*. New Orleans Louisiana USA: ACM, Oct. 1999, pp. 147–153.
[39] "Doctest — Test interactive Python examples — Python 3.11.3 documentation," https://docs.python.org/3/library/doctest.html.
[40] "Use JSDoc: Getting Started with JSDoc 3," https://jsdoc.app/about-getting-started.html.
[41] "Rust-lang/cargo: The Rust package manager," https://github.com/rust-lang/cargo.
[42] "Docs.rs," https://docs.rs/.
[43] "Move_items_with_progress in fs_extra - Rust," https://docs.rs/fs_extra/1.3.0/src/fs_extra/lib.rs.html#601-774.
[44] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, Nov. 2020.
[45] "The Rust Standard Library," https://doc.rust-lang.org/std/.
[46] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 246–257.
[47] "Trust-dns-proto - crates.io: Rust Package Registry," https://crates.io/crates/trust-dns-proto.
[48] "Mistodon/config_struct: Generate structs at compile time from arbitrary config files." https://github.com/mistodon/config_struct.
[49] "Documentation mismatch for WaitTimeoutResult::timed_out," https://github.com/rust-lang/rust/issues/110045.
[50] "Fix the example in document for WaitTimeoutResult::timed_out," https://github.com/rust-lang/rust/pull/110056.
[51] "Documentation mismatch for primal_check::miller_rabin · Issue #56 · huonw/primal," https://github.com/huonw/primal/issues/56.
[52] "Rust Subreddit," https://www.reddit.com/r/rust/.
[53] "The Rust Programming Language Forum," https://users.rust-lang.org/.
[54] "Rust-lang/rust-clippy: A bunch of lints to catch common mistakes and improve your Rust code." https://github.com/rust-lang/rust-clippy.
[55] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, May 2020.
[56] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1385–1397.
[57] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A Transformer-based Approach for Source Code Summarization," May 2020.
[58] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Paul, "SeTransformer: A Transformer-Based Code Semantic Parser for Code Comment Generation," *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 258–273, Mar. 2023.
[59] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*. San Francisco, CA, USA: IEEE, May 2013, pp. 83–92.
[60] M. Geng, S. Wang, D. Dong, S. Gu, F. Peng, W. Ruan, and X. Liao, "Fine-Grained Code-Comment Semantic Interaction Analysis," p. 12, 2022.
[61] M. Vidoni, "Understanding Roxygen package documentation in R," *Journal of Systems and Software*, vol. 188, p. 111265, Jun. 2022.
[62] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Montreal, QC, Canada: IEEE, Apr. 2012, pp. 260–269.
[63] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 112–122.
[64] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete TODO comments," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 218–229.