Towards Understanding Rust Documentation at an Ecosystem Scale

***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 101,868 crates from the Rust crate registry, `crates.io`, to conduct a quantitative study to investigate the presence, completeness, size, and inconsistency of documentation in the Rust ecosystem, 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) the proportion of crate homepages in the Rust ecosystem that include documentation is 44.31%; 2) the Rust ecosystem's documentation ratio is very low, with 37.24% of crates having no documentation and only 13.23% of public items having documentation; and 3) the main causes of document-code inconsistency are unsynchronized updates of document code and errors caused by copy/paste operations. 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 in the early stages of development. 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 that allows developers to write Rust testing code within documentation. During testing, the Rust testing code in documents is 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 dual 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 [3] [4].

To better guide the use of documentation, Rust provides an official Rustdoc specification [5] as well as a community guideline [6], proposing three important principles that 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 a Rust crate, such as traits, structs, enums, functions, methods, macros, and type definitions, should have an explanation of its 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` [7], 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 confusion [8], inconsistencies [9] [10], vulnerabilities, or even bugs [11] [12] [13] [14].

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 [15] [16] [17] [18]. 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 *non-mandatory*. Therefore, Rust does not provide any official checking tool distributed with its official compiler `rustc` [19]. 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 [15] [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 does the documentation of Rust programs adhere to the community guidelines? What is the size of documentation that Rust developers write? What is the proportion of document-code inconsistencies in the 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, size, and inconsistency, in a fully automated manner.

Second, we created three datasets with 101,868 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 the challenges they encounter.

The empirical results give interesting findings and insights, such as: 1) the proportion of crate homepages in the Rust ecosystem that include documentation is 44.31%; 2) the Rust ecosystem's documentation ratio is very low, with 37.24% of crates having no documentation and only 13.23% of public items having documentation; and 3) the main causes of document-code inconsistency are: unsynchronized updates of document code and errors caused by copy/paste operations.

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 start writing documentation in the early stages of development and pay more attention to document testings.

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 potential document-code inconsistencies.

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.10050847`.

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. Sections 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. It was initially designed by Graydon Hoare in 2006 and was first publicly released in 2010 [24]. 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. The ownership and lifetime are both checked and enforced

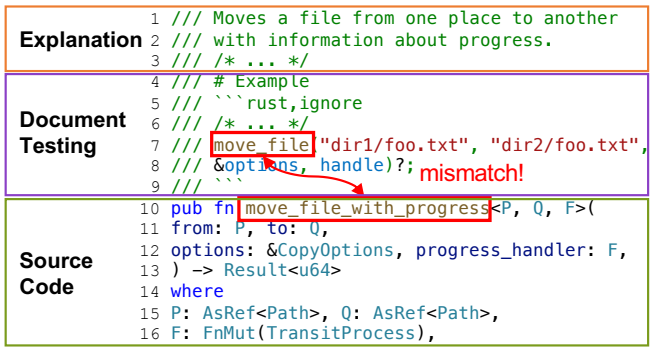


Fig. 1: A motivating example.

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, the 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 [5].

On the other hand, Rust provides specific documentation methods and a simple-to-use tool called `rustdoc`, which generates user-friendly HTML documents. Furthermore, to simplify package building and management, Rust provides the `cargo` tool [41] and eco-friendly package repositories `crates.io` [7] and `Docs.rs` [42]. Consequently, Rust's documentation is comprehensive and effective, making the learning curve for Rust programmers less steep.

Specifically, Rust offers a novel feature called document testing which enables programmers to write Rust code within documentation that 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 comprise a short description of the function, referred to as "explanation", which supports **H2**; lines 4 to 9 constitute a runnable example, denoted as "document testing", which supports **H3**; lines 10 to 16 display the source code.

However, the document testing is flawed due to the 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, thus will not be executed by the test tool `rustdoc`; 2) if the function `move_file` does exist, the tests will erroneously invoke and test the *wrong* function.

III. METHODOLOGY

In this section, we present the research questions (§ III-A) that guide our study, as well as the data selection criteria (§ 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 **H1** holds.

RQ2: Completeness. To what extent does the documentation of Rust programs adhere to the community guidelines?

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 **H2** and **H3** hold.

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

Previous research [4] has demonstrated that redundant documentation and duplicated content are significant factors that affect the maintainability of documents. These findings provide the motivation for exploring RQ3, which focuses on estimating document size.

RQ4: Inconsistency. What is the proportion of document-code inconsistencies in the 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 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?

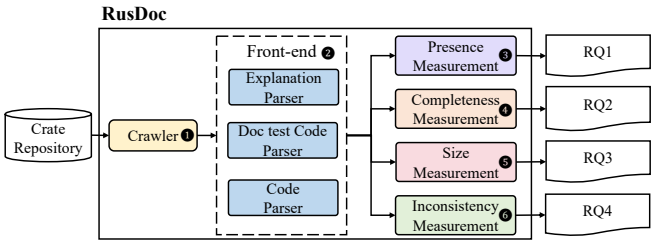


Fig. 2: RUSDOC architecture.

The motivation for RQ5 was to investigate whether the introduction of Rustdoc guidelines [6] 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 about 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, to cover as wide a range as possible, we included as many Rust crates as possible in our study. We selected the latest versions of 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 for 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 (§ IV-A), and its architecture (§ IV-B). We then discuss the design and implementation of the crawler module (§ IV-C), the front-end module (§ IV-D), the presence (§ IV-E), completeness (§ IV-F), size (§ IV-G), and the inconsistency measurement module (§ IV-H), 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 through manual code inspection.

Second, the study can be applied to any Rust projects with different structures, rather than being limited to specific ones. RUSDOC is designed with the principles of modularity and extensibility, making it 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 six 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 the Rust source code as input 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.

Finally, the presence (❸), completeness (❹), size (❺), and inconsistency (❻) measurement modules take the result of the front-end module as input and answer **RQ1**, **RQ2**, **RQ3**, and **RQ4** respectively.

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 the source code, we also collect corresponding metadata, such as the 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. We then proceed to download the data from the crate repository, following the crawler policy provided by it. After fetching the data, we set up a database of Rust crates that stores comprehensive metadata, including the crate's source code and version details. This database will be utilized in 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 relevant 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. 3) AST generation: the document testing code parser and the code parser take the Rust code as input, and build the Rust abstract syntax

trees (AST). The AST is a tree representation of the source programs, particularly containing 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 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 actually exists. To this end, RUSDOC incorporates a presence measurement module to provide a comprehensive assessment of the documentation in the Rust ecosystem, including whether a crate's homepage has documents and the proportion of public items of each crate. The assessment is further used to answer **RQ1**.

To analyze the presence, we took a two-pronged approach. First, to validate the veracity of **H1**, we examine the inclusion of corresponding documents on each crate's homepage. The presence of documentation on a crate's homepage is crucial as it offers a concise overview of the crate and showcases usage examples. These statistics enable us to visually assess the adherence of Rust library developers to community guidelines.

Second, we computed the documentation ratio for each crate by counting the number of public projects (#pub) and the number of public projects with accompanying documentation (#doc), subsequently calculating $\frac{\#\texttt{doc}}{\#\texttt{pub}}$. We believe that the documentation ratio reflects how well a crate is documented, and that crates with high documentation ratios are better maintained in comparison.

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**.

First, to verify **H2**, we calculate the documentation ratio on each public feature (module, trait, struct, enum, function, method, macro, and type definition) in the dataset. Note that this indicator counts all public items in the dataset, grouped by features. Specifically, we count the number of documented features $\#\texttt{feature}_{\texttt{doc}}$, the number of public features $\#\texttt{feature}_{\texttt{pub}}$, then we calculate $\frac{\#\texttt{feature}_{\texttt{doc}}}{\#\texttt{feature}_{\texttt{pub}}}$.

Second, to verify **H3**, we separately count whether there is an explanation and a document testing on each *documented item*. We count the items with explanations #explanation, the items with document testings #testing, and the items with either or both of them #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. Size Measurement

To answer **RQ3**, we introduce a size measurement module that measures the size of a document by taking as input the results output by the explanation parser.

To measure the size of a document, an appropriate method must be employed. Counting the number of lines per document is a viable option. Although the subjective nature of this metric has been acknowledged in previous research [44], we contend that it does, to some extent, reflect the human effort invested in the document.

Since 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.

H. 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**.

Detecting inconsistencies between documentation and code poses a significant challenge. Prior work [15] [45] [14] has proposed rule-based approaches to identify pre-existing inconsistencies within specific domains. However, this does not apply to the detection of document testing. As a first step, we performed a detection of document-code inconsistencies for functions and methods. More specifically, if a function or method is not referenced in its respective document testing, it is considered an inconsistency. Since this does not meet user expectations, as mentioned in previous work [4], most developers pointed out that incorrect code examples are an important issue affecting the correctness of the documentation. For example, we can detect the inconsistency in Fig. 1 even though it uses the 'ignore' attribute.

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

To further delve into the underlying reasons behind document-code inconsistencies, we conduct a manual review of select cases. The manual review consists of two steps. First, we investigate the commit history of the code repository to ascertain whether errors arise due to the documentation not being updated in sync with the code. Second, we consult developers regarding the potential causes of specific examples.

V. EMPIRICAL RESULTS

In this section, we present the empirical results by answering the research questions. We first illustrate the experimental

TABLE I: Dataset used in this study.

Name	Source	Size
DS1	crates.io [7]	101,868 crates
DS2	crates.io	500 crates
DS3	Rust standard library [46]	404 files

setup (§ V-A), then present the datasets (§ V-B), and then answer the previously mentioned research questions (§ V-C to § V-H).

A. Experimental Setup

All the experiments and measurements were conducted on a server with one 4 physical Intel i7 core CPU (8 hyperthreads) and 12 GB of RAM, running on Ubuntu 21.04.

B. Datasets

We created three datasets, as shown in Table I. Our data selection criteria are presented in Section III-B. We hereby present the actual crates incorporated in our study and elucidate the reasons for not including all Rust crates in our dataset.

First, we selected the latest versions of 101,868 crates on the Rust community's crate registry (crates.io). Out of the total 127,232 crates on crates.io, we excluded 4,711 crates whose latest versions were not downloaded successfully. As the aim was to analyze the quality and inconsistency of the crate documentation, we excluded crates that were compiled into binary (20,653 in total). Afterwards, our dataset DS1 contains 101,868 crates, which represents 80% of the total registered crates.

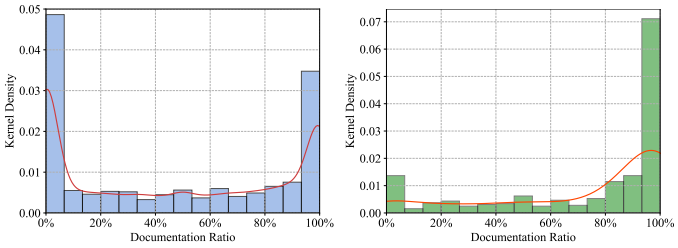
Second, we analyzed the popular and actively-maintained crates in our dataset. Considering the criterion utilized in prior work [47], we opted to employ the download count as an indicator of popularity. This choice is justified by the fact that the download count accurately reflects both the extent of user engagement and the attention directed towards the crate. For this analysis, we selected 500 crates whose downloads accounted for 74% of the total downloads. 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 applied RUSDOC to the dataset DS1. Our findings revealed that out of all the library crates, 45,136 crates (44.31%) have documents on their crate homepage. This suggests that a significant proportion of crate maintainers in the Rust ecosystem prioritize documentation and are dedicated to enhancing the user experience.

Subsequently, we computed the distribution of documentation ratios for DS1 and DS2. Fig. 3 displays the histogram and kernel density estimation curve for both datasets. Upon examining the figure, it became evident that in DS1, most crates either have zero or all documents, while the documentation ratios of other crates are evenly distributed. Conversely, DS2 has a higher concentration of crates with a 100% documentation ratio. We further listed the number and proportion of crates in DS1 with no documents, some documents, and all documents in TABLE II. Overall, 62.76% of crates contained at least one document, while 37.24% of crates had no documentation at all.



(a) DS1 (b) DS2

Fig. 3: Document ratio distribution and kernel density estimation of DS1 and DS2, respectively.

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

Document	#Crates	Ratio
None	37,932	37.24%
Some	63,936	62.76%
All	16,650	16.34%

Summary: The proportion of crate homepages in the Rust ecosystem that include documentation is 44.31%, partially supporting RustDoc hypothesis **H1**. Additionally, it is observed that 37.24% of crates have no documentation.

Suggestion: We recommend that developers of Rust libraries place a priority on creating comprehensive homepage documentation for their crates, to assist users in understanding and effectively utilizing them. These results further emphasize the potential and necessity of automated documentation generation techniques.

D. RQ2: Completeness

To answer RQ2 by investigating the completeness of the Rust documentation, we first applied RUSDOC to the dataset 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 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 DS1, DS2, and DS3, respectively.

We then investigated the ratio of explanation and document testing in DS1. The empirical results are presented in TABLE IV. The first column lists public features. The second column

TABLE III: Documentation ratio for each public feature.

Feature	DS1	DS2	DS3
Module	16.34%	57.31%	78.16%
Function	36.13%	50.85%	25.49%
Struct	15.50%	21.22%	34.51%
Enum	20.15%	26.60%	67.74%
Trait	**51.73%**	**78.82%**	**92.55%**
Method	13.82%	69.86%	36.31%
Macro	51.48%	62.75%	80.00%
Typedef	**2.90%**	**2.12%**	**5.30%**
Average	**13.23%**	**34.20%**	**34.00%**

indicates the ratio of explanation in all documented items, respectively. The third column indicates the ratio of document testing.

The empirical results provide four interesting findings and insights: 1) the documentation ratio of all public items is 13.23% in all datasets, which means that more than half of the items do not have documents. However, the documentation ratio of DS2 (34.2%) and DS3 (34.00%) is significantly higher than that of DS1 (13.23%); 2) traits have the highest ratio (51.73% in DS1), while type definitions have the lowest ratio (2.90% in DS1); 3) the explanation ratio of all items is higher than 99%, meaning that most documents contain a description in natural language. This partially supports Rustdoc's second hypothesis (**H2**); and 4) except for functions (13.24%) and macros (41.12%), the document testing ratio of all items is less than 10%, indicating that most items do not contain document testings. This violates the third Rustdoc hypothesis (**H3**).

We then speculate on three possible reasons for these results. First, trait is a Rust feature that specifically outlines how different types should behave and interact. Trait users are more likely to consult the documentation, while 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>;
```

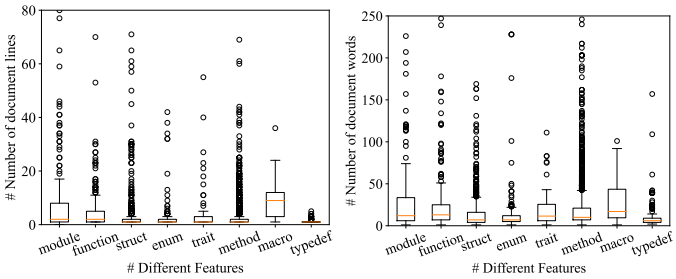
which means Result<T>is a type alias of Result<T, Error>.

Secondly, users often require sample code for functions and macros as they may be unclear on how to use them. On the other hand, information related to structs and enums is self-explanatory, and users can directly access details from their fields without the need for document testing.

Finally, we hypothesized that the lack of documentation may be due to subjective reasons on the part of the developers. To confirm this hypothesis, we conducted a questionnaire survey in **RQ6** and obtained answers to this question.

TABLE IV: Explanation and document testing ratio for documented public features, in DS1.

Feature	Explanation	Document Testing
Module	99.99%	7.10%
Function	99.92%	**13.24%**
Struct	99.99%	5.28%
Enum	99.99%	0.93%
Trait	99.99%	5.70%
Method	99.98%	6.07%
Macro	99.76%	**41.12%**
Typedef	99.99%	0.58%
Average	99.98%	5.54%



(a) Document lines (b) Document words

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

Summary: The documentation in the Rust crate repository is inadequate (13.23%), Among them, traits have the highest documentation ratio (51.73% in DS1), and type definitions have the lowest ratio (2.90% in DS1). The Rust ecosystem has the explanation ratio of 99.98% for documented items, which partly supports the second Rustdoc hypothesis (**H2**). However, the repository has a low ratio of document testings (5.54%), which violates the third Rustdoc hypothesis (**H3**).

Suggestion: We recommend that maintainers of libraries increase the document testing efforts, as this practice can significantly enhance the readability and usability of the documentation. This challenge can be addressed by developing automatic generation techniques specifically for Rust document testings.

We also recommend that Rust's documentation guidelines more clearly define when documentation is required and when it can be omitted. For example, according to the empirical findings of DS3 (which represents the best practice for documenting Rust), traits necessitate the most documentation, whereas type definitions require the least to some extent.

E. RQ3: Size

To answer RQ3 by investigating the size of the document, we apply RUSDOC to the dataset DS1. The boxplots in Fig. 4 show the distribution of document lines 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. This suggests that Rust crate documentation, on average, tends to be concise

and straightforward. Through further automated inspection, we found that a large proportion (i.e. 66.22%) of small documents only occupy 1 line and have a simple one-sentence description of the corresponding code.

Second, macros have the largest documentation size, which may indicate that macros generally require more detailed and extensive explanations to clarify their functionality and usage. The definition type has the smallest documentation size, which means that the type definition may be more self-explanatory or require less documentation due to its inherent clarity.

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.

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 provide two interesting findings and insights. First, among all crates with document testing on dataset DS1, the overall inconsistency ratio is 3.51%, while the inconsistency ratio for dataset DS2 is 1.81%. Second, 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 the 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 a misunderstanding of the code, which may further lead to bugs or vulnerabilities. We have reported this issue [49] to the developers. A developer has fixed the bug, and the fix has now been merged into the Rust master branch [50].

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

Ratio	2018	2019	2020	2021	2022	2023
P[1]	6.48%	7.09%	7.79%	7.79%	12.77%	13.23%
C[2]	10.29%	10.12%	10.10%	10.08%	5.94%	5.54%
I[3]	2.01%	2.57%	2.78%	2.53%	2.70%	3.51%

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

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:

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

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

Summary: The inconsistency ratio between documentation and code is 3.51%. Two primary underlying causes contribute to inconsistencies: document-code desynchronization and errors arising from copy/paste actions.

Suggestion: We recommend that the maintainers of the library publish the source code on code-sharing platforms (e.g., GitHub), thereby encouraging and enabling external contributors to add new content or propose fixes for the documentation. We also recommend the development of a just-in-time inconsistency detection tool for document testing.

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 testing of these public items. The inconsistency ratio represents the number of document testing mismatches #`mismatch` divided by the number of all document testings #`total`, i.e. $\frac{\#\texttt{mismatch}}{\#\texttt{total}}$.

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%

The empirical results give three interesting findings and insights. First, the presence of documentation increased from 6.48% in 2018 to 13.23% in 2023, and we speculate that this increase is due to developers paying more attention to documentation, and 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 3.51% in 2023.

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 13.23%) and inconsistency ratio (from 2.01% to 3.51%) of documents have increased, and the complete ratio has decreased from 10.29% to 5.54%. 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 designed a survey and posted it on the Rust Subreddit [52] and Rust user forum [53], a selection used in prior work [47], collecting data from 30 respondents (as of this study). The participants were asked about the issues they encountered while writing and consulting documentation.

The first question (**SQ1**) asked whether or not Rust developers add document testings to their crates. The majority (72%) responded that they added a few, while a minority (28%) stated that they added none or all, aligning with the findings for **RQ1** and **RQ2**.

The second question (**SQ2**) asked Rust developers to indicate the reasons for not including testing in the documentation. More than half of the respondents (63%) reported insufficient time as the main factor, while some of them (53%) mentioned either adding it or planning to do so in the future. Concerns about slow execution time or ineffective testings of certain aspects through examples were expressed by 8% of the respondents. They also considered utilizing untested examples to be poor practice. Other reasons selected include the belief that the API's clarity was satisfactory without the need for an example (38%) and a lack of knowledge on how to add it (8%).

The third question (**SQ3**) focused on the difficulties encountered when writing the testing. The majority of respondents (63%) reported a lack of understanding of how to write and test effective examples in Rust. Other challenges mentioned were the potential deceleration of testing processes and the absence of support or feedback from official tools (such as clippy [54], rustfmt, and rust-analyzer) as well as third-party tools. The responses to **SQ2** and **SQ3** provide further insights into the underlying causes investigated in **RQ2**.

The fourth question (**SQ4**) inquired about the challenges faced when consulting other crate documentation. TABLE VI shows the typical responses to this question, including the encountered problem, a short description of the problem, and the percentage of respondents experiencing it. As this question allowed for multiple responses, participants could select more than one option.

Finally, **SQ5** solicited suggestions for improving Rust documentation. It is worth noting that this question was optional, and not all participants answered it. Here are a few some representative answers for this question, which may not encompass 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%).

Suggestion: We recommend starting documentation in the early stages of the software lifecycle and designating adequate time, effort, and resources for documentation within the project. For developers who don't know how to get started, we recommend consulting the official Rustdoc book [23], which offers valuable guidance and examples.

VI. Implications

This paper presents the first and most 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 guidelines should provide clearer guidance on when documentation is required

and when it can be omitted; and 2) the tool can be utilized to conduct completeness and inconsistency documentation checks before they are published to `crates.io`.

For tool builders. We recommend that tool builders develop corresponding tools to detect real-world document-code inconsistencies, especially for documentation testing. Since our prototyping system took the first step toward it, it was feasible to develop such a tool. We advocate for the advancement of automatic documentation (e.g., [55] [56] [57]) and sample code generation technology as solutions for addressing the issue of inadequate documentation.

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 testing in the documentation became more common. Based on the observation, we suggest that Rust developers should: 1) start documentation in the early stages of the software lifecycle and clearly allocate time, effort, and resources to documentation within the project; 2) create crate homepage document and more document testings to improve document usability, refer to the rustdoc book if necessary; and 3) put the code repository on code-sharing platforms for feedback and external contributions.

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.

Dataset. 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] [47]. 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.

Dynamic inconsistency detection. Our results mainly focused on the 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 to future work to study document-code inconsistency, by leveraging prior work in this direction [21] [22].

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. However, the work in this study represents a novel contribution to this field.

Documentation studies. Numerous research efforts focused on analyzing documentation. Steidel et al. [58] adopted machine learning methods to classify comments into seven categories, and further analyzed and evaluated the quality of code comments. Geng et al. [59] proposed Fosterer as an automated solution to analyze the semantic relationship between code and specific tokens in comments. Vidoni [60] mined and analyzed 379 systematically selected open-source R packages for their quality in terms of documentation existence, distribution, and completeness. However, none of these studies conducted a large-scale empirical analysis of the Rust ecosystem.

Document-code inconsistency studies. Document-code inconsistency has been studied extensively. Tan et al. [15] proposed iComment, a technique using NLP, machine learning, and program analysis to detect code-comment inconsistencies in locking mechanisms. Then they [45] further proposed @TCOMMENT, an approach for testing Javadoc comments, specifically focusing on method properties related to null values and exceptions. JavadocMiner [16] was designed to evaluate the quality of Java comments and code-comment consistency through a set of heuristics. Ratol et al. [61] proposed Fraco, to detect inconsistencies in code comments due to renamed refactoring operations performed on identifiers. TDCleaner [62] was designed to automatically identify and remove obsolete TODO comments in software projects. However, none of these studies can be directly applied to investigate Rust document testing, due to the feature discrepancies between Rust and these programming 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 called RusDoc, we conducted 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 the low ratio of documentation, insufficient document testing, and document-code inconsistency. Additionally, we also surveyed Rust developers to understand their perceptions of document testing and the challenges they face. This work represents a first step towards understanding Rust documentation culture among the Rust community, emphasizing the need to further strengthen the Rust ecosystem and establish better documentation guidelines.

References

[1] "Rust Programming Language," https://www.rust-lang.org/.

[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] E. Aghajani, C. Nagy, O. L. Vega-Marquez, M. Linares-Vasquez, L. Moreno, G. Bavota, and M. Lanza, "Software Documentation Issues Unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1199–1210.

[4] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: The practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 590–601.

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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)*. Buenos Aires: IEEE, May 2017, pp. 27–37.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[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] 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.
[46] "The Rust Standard Library," https://doc.rust-lang.org/std/.
[47] 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.
[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-clippy," 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. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A Transformer-based Approach for Source Code Summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4998–5007.
[58] 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.
[59] M. Geng, S. Wang, D. Dong, S. Gu, F. Peng, W. Ruan, and X. Liao, "Fine-grained code-comment semantic interaction analysis," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 585–596.
[60] M. Vidoni, "Understanding Roxygen package documentation in R," *Journal of Systems and Software*, vol. 188, p. 111265, Jun. 2022.
[61] 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.
[62] 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.