Assessing the Alignment between the Information Needs of Developers and the Documentation of Programming Languages: A Case Study on Rust

FILIPE ROSEIRO COGO, Centre for Software Excellence, Huawei, Canada
XIN XIA, Software Engineering Application Technology Lab, Huawei, China
AHMED E. HASSAN, School of Computing, Queen's University, Canada

Programming language documentation refers to the set of technical documents that provide application developers with a description of the high-level concepts of a language (e.g., manuals, tutorials, and API references). Such documentation is essential to support application developers in effectively using a programming language. One of the challenges faced by documenters (i.e., personnel that design and produce documentation for a programming language) is to ensure that documentation has relevant information that aligns with the concrete needs of developers, defined as the missing knowledge that developers acquire via voluntary search. In this article, we present an automated approach to support documenters in evaluating the differences and similarities between the concrete information need of developers and the current state of documentation (a problem that we refer to as the topical alignment of a programming language documentation). Our approach leverages semi-supervised topic modelling that uses domain knowledge to guide the derivation of topics. We initially train a baseline topic model from a set of Rust-related Q&A posts. We then use this baseline model to determine the distribution of topic probabilities of each document of the official Rust documentation. Afterwards, we assess the similarities and differences between the topics of the Q&A posts and the official documentation. Our results show a relatively high level of topical alignment in Rust documentation. Still, information about specific topics is scarce in both the Q&A websites and the documentation, particularly related topics with programming niches such as network, game, and database development. For other topics (e.g., related topics with language features such as structs, patterns and matchings, and foreign function interface), information is only available on Q&A websites while lacking in the official documentation. Finally, we discuss implications for programming language documenters, particularly how to leverage our approach to prioritize topics that should be added to the documentation.

CCS Concepts: • **Software and its engineering** → *Software creation and management;*

Additional Key Words and Phrases: Documentation, programming languages, Rust, Q&A websites, Stack Overflow, RustForum, topic models, domain knowledge

ACM Reference format:
Filipe Roseiro Cogo, Xin Xia, and Ahmed E. Hassan. 2023. Assessing the Alignment between the Information Needs of Developers and the Documentation of Programming Languages: A Case Study on Rust. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 43 (March 2023), 48 pages.
https://doi.org/10.1145/3546945

Authors' addresses: F. R. Cogo, Centre for Software Excellence, Huawei, Canada; X. Xia, Software Engineering Application Technology Lab, Huawei, China; A. E. Hassan, School of Computing, Queen's University, Canada.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1049-331X/2023/03-ART43 $15.00
https://doi.org/10.1145/3546945

1 INTRODUCTION

Programming language documentation is the set of technical documents that describe high-level concepts of a programming language and allow application developers to learn and use the language in an effective way [99]. This documentation is officially maintained and distributed and typically includes a description of the language concepts, reference manuals, usage examples, and tooling tutorials. Examples of such a documentation include the Java Documentation,[1] the Rust Documentation,[2] and the Go Documentation,[3] among others. Without a sufficient set of documents, both novice and experienced developers lack the necessary information about using a programming language to address their needs and improve their knowledge.

The documentation of a programming language needs to be periodically updated and to follow along with the language evolution. To produce informative documents and to plan the allocation of resources to documentation-related activities, documenters (i.e., personnel devoted to writing documentation) should be able to identify the most important and practical topics that developers need and to compare such topics against the current version of the documentation [82]. Documenters should also be able to identify topics of the documentation that are not well covered and prioritize topics that should be improved. Similarly, documenters should be able to identify sufficiently documented topics (in relation to developers' information needs) to avoid redundancies in the maintained documentation. We define concrete information needs as the missing knowledge that is acquired via the active process of asking other developers. We refer to *topical alignment* of the documentation of a programming language as the difference between the topics that make up the concrete information needs of developers and the covered topics by the current state of the documentation. Maximizing the topical alignment of the produced documentation is not a trivial task, as documenters are not always aware of the concrete information needs of developers.

Complete and accurate documentation is of fundamental importance in providing developers with information about the usage of a programming language. Despite its importance, assessing the topical alignment of the documentation of a programming language is an open research problem [82, 92]. Typically, communities and sponsors around different programming languages employ expensive and time-consuming survey methods to assess the topical alignment of their documentation [1, 4, 9]. In this article, we describe a machine learning–based approach to tackle this important problem. Although our approach is not supposed to replace surveys that are typically conducted, it can be leveraged by programming language documenters to assist in the design and complement the results of such surveys. The main idea behind our approach is to build two topic models—one that represents the information needs of developers and another one that represents the covered topics by the current documentation of the language—and then assess the similarity and dissimilarities between the topics of the two models. To build such models, we use semi-supervised topic modelling [64] that leverages domain knowledge, represented by a set of associated keywords with each topic (a.k.a. *anchor words*), to guide the derivation of latent topics from textual information [49, 51, 64, 89].

We demonstrate the practical application of our approach by empirically assessing the topical alignment of the Rust official documentation. Rust is a programming language that is steadily gaining the attention of software engineering practitioners, being voted as the "most loved" language by developers for over six consecutive years [10]. Nonetheless, when participants of the annual Rust Survey [7] are asked about what the Rust community can do to improve Rust's adoption, respondents frequently answer that training and documentation are two essential efforts. Therefore,

[1] https://docs.oracle.com/en/java/.
[2] https://www.rust-lang.org/learn.
[3] https://golang.org/doc/.

beyond demonstrating the application of our approach, an empirical study of the topical alignment of Rust documentation provides practical recommendations to an important community of documenters.

For our case study, we encode the Rust domain knowledge into 47 different ***knowledge units* (KUs)** that are induced from a manual categorization of the language's official documentation [22]. A KU represents a set of related concepts that a skilled Rust developer must know. Each KU is associated with one topic that our semi-supervised topic model derives. Each KU is also associated with a set of anchor words (e.g., the "primitive type" KU is associated with anchor words such as "tuple" and "array") that are given as input to the semi-supervised topic models. We then build a *concrete model* from the data of two popular **Question & Answer (Q&A)** websites [54, 66, 91, 95] where developers actively seek for information about Rust [58, 93], namely, Stack Overflow and Rust Forum (the officially maintained Q&A website by the Rust community). In addition, we build a *documentation model* from the data of the official Rust documentation.

For both the concrete and documentation models, we analyze the *prevalence of each KU* (i.e., the number of documents in which a KU occurs). Based on the prevalences, we identify the *convergent KUs* (i.e., the KUs that are highly prevalent in both the concrete and documentation models), the *divergent KUs* (i.e., the KUs that are highly prevalent in the concrete model but not in the documentation model), and the *absent KUs* (i.e., KUs with a low prevalence in both models). We also analyze the *awareness of each KU* (i.e., the number of answers that each KU receives in the Q&A posts that are associated with the KU). Based on these analyses, we answer three research questions (RQs):

(RQ1) How the prevalence of KUs differs between the concrete information needs of developers and the Rust documentation?
In general, there is a strong agreement between the concrete information needs of developers and the contents of the official Rust documentation. Still, specific KUs that are associated with many Q&A posts have a low coverage by the Rust documentation. We also observed KUs that are seldomly discussed in Q&A websites and also have a low coverage by the official documentation.

(RQ2) How many KUs are absent, divergent, or convergent in the Rust documentation?
We identified 10 absent, 6 divergent, and 10 convergent KUs in terms of the frequency of occurrence, half of the absent KUs categorized as a KU for programming niches (e.g., network, game, and database development). In addition, one-third of the KUs that are categorized as language features (e.g., structs, patterns and matchings, and foreign function interface) are divergent.

(RQ3) Do convergent, divergent, and absent KUs exhibit different awareness in Q&A websites?
In general, absent KUs have higher ranks of attention (i.e., number of answers in Q&A posts) and lower ranks of agreement (i.e., the odds of having an accepted answer), suggesting KUs that should be prioritized by documenters (e.g., logging and command line arguments parsing). Conversely, convergent KUs have lower ranks of attention and higher ranks of agreement.

The main contributions of our article are:

- *Methodological*: We propose an approach to assess the topical alignment of the documentation of a programming language using semi-supervised topic models of the posts of Q&A websites and a language's documentation. To foster further research in the area of topical alignment of programming languages, we also provide supplementary material that contains our data and trained model [30].
- *Technical*: We perform an empirical study about the topical alignment of the Rust documentation, followed by a discussion of the implications of our findings for documenters of programming languages. In our empirical study, we develop a set of metrics and

visualizations to assess the topical alignment of the documentation of a programming language. Our empirical study also identifies topics that lack coverage from the Rust documentation and which of such topics should be prioritized by documenters.

- *Conceptual*: We modelled the domain knowledge of Rust through a careful analysis of the official documentation that is maintained by the community. Our domain knowledge is validated by a specialist with years of contribution to the analyzed material (as one of the core contributors of Rust documentation). Our domain knowledge is encoded as anchor words and it can be used by a variety of semi-supervised topic modelling techniques that adopt this knowledge representation format [49, 51, 64, 89].

Article Organization: Section 2 motivates our study. Section 3 provides background material. Section 4 presents our data collection procedure. Section 5 describes our results. Section 6 discusses the implications of our findings for documenters. Section 7 reflects on the threats to the validity of our study. Section 8 presents our conclusions.

2 MOTIVATION

It is common for communities in charge of maintaining programming languages to release surveys to understand the users' opinions about diverse aspects of the language. By analyzing the latest surveys from five of the "most loved" languages[4] [10] (in order, Rust, Python, Kotlin, Go, and Julia), we found that users mention the importance of documentation in all of these surveys. In the Rust survey, participants (n = 8,323) were asked about what they think could be done to improve the adoption of Rust, and the largest category of answers was "documentation and training" [7]. In the Python survey, 14% of the respondents (n > 28,000) pointed "clear documentation" as the answer to an open question about their three favorite features of the language [6]. The Kotlin survey (n = 1,163) showed that almost 25% of the respondents use the language's official documentation to set up a project. One of the conclusions drawn by the Kotlin community based on their survey is that "*to increase the level of satisfaction with Kotlin Multiplatform, we need to continue to improve and maintain the documentation*" [18]. Participants of the Go survey (n = 9,648) were asked about what could be done to make the Go community more welcoming, and 21% of the respondents of this question (n = 275) pointed to "improvement of learning resources and documentation" [4]. In addition, other 62% of the respondents (n = 2,476) pointed that they struggle to "find enough information to fully implement a feature of my application" in the official Go documentation. Similarly, in the Julia survey (n = 2,565), 26% of the respondents said that "insufficient documentation" is one of the biggest non-technical problems with the language [11].

The survey responses show that documentation quality is paramount in promoting the adoption and the usage of a programming language. Moreover, communities devoted to the development of a programming language are, in general, willing to improve documentation. Also, the official documentation occasionally lacks information, compelling developers to find alternative forms to obtain information. Therefore, documenters of programming languages will benefit from new approaches to understand the state of the maintained documentation, especially if these approaches can complement the results and potentially reduce the costs of designing surveys. As the development of the official documentation of a programming language typically depends on a centralized team and follows a push-based distribution model (i.e., documenters are responsible for the definition of the available information), each update to the documentation contents needs to be carefully considered. In particular, to maximize the utility of the documented information, documenters want to assess the alignment between the maintained documents and the information

[4]We did not find any survey for the TypeScript language, featured second on the rank of "most loved" languages.

needs of developers. This requirement is exemplified in the surveys mentioned above. For example, one of the questions in the Go survey asks "how helpful is official Go documentation for achieving your programming goals" [4]? Also, one of the immediate actions stated by the Kotlin community in response to the results of their documentation-related questions includes "*focus (...) on common user scenarios that are not well covered in documentation*" [18].

In this article, we describe a machine learning-based approach that documenters can adopt to assess the alignment between the concrete information needs of developers and a programming language's documentation. The information needs from developers are represented by the topics derived from Q&A websites, as this type of documentation follows a pull-based distribution model (i.e., information seekers are responsible for the definition of the available information), and developers post legitimate questions to obtain voluntary answers from other developers. Our approach, described in detail in Section 4.1, uses semi-supervised topic modelling that leverages domain knowledge to cluster documents into topics (each topic associated with one KU from Rust). We first build a concrete model using data from rust-related Q&A posts, then we use this model to derive the topics of the official Rust documentation. A set of KU metrics, described in detail in Section 4.2, are extracted from the built models and compared against each other, providing documenters with information about the similarities between the KUs of the concrete and the documentation models.

3 BACKGROUND AND RELATED WORKS

This section discusses the background material of our study, as well as related works. Section 3.1 presents prior studies about programming language documentation and software documentation in general. Section 3.2 presents prior works that extracted developer's information needs from Q&A websites. Section 3.3 presents a semi-supervised topic modelling technique called Anchored **Correlation Explanation (CorEx)** and discusses the advantages of applying this specific technique over unsupervised techniques to mine textual data.

3.1 Programming Languages Documentation for Software Engineering

In the context of software engineering, programming language documentation refers to the technical information that communicates to developers concepts about the *usage* of a programming language (i.e., high-level concepts), in contrast with its *implementation* (i.e., formal definitions) [86]. Examples of such documents include manuals, training material, and API references. The maintenance of programming language documentation is deemed as a fundamental practice since the initial days of software engineering. In 1963, referring to the documentation of the IPL-V programming language (released a few years earlier), Allen [79] wrote that "*the main documentation of IPL-V can be considered complete, official and almost permanently fixed*" and that the concern of future updates to the language documentation was addressed: "*(...) documentation is kept on tape with procedures for updating and modifying the documentation.*" Yngve et al. [99] described some of the pioneering research in programming languages documentation, recognizing the importance and discussing the challenges of maintaining this type of documentation. Interestingly, more recent research recognizes (general) documentation maintenance as an often overlooked software engineering practice [69, 73], pointing to the cost of maintaining documentation as one of the drivers of this scenario.

At the intersection of programming languages documentation and software engineering automation, researchers proposed different approaches to automatically generate documentation, often by analyzing the structure of source code. Such automated approaches aim to supporting different development tasks (such as exception handling [56]) and different niches (such as scientific programming [78]). Automated approaches are also used to generate documentation to different

features of the language, such as its API [76, 87, 100], components [77], and specific syntatic expressions [46]. One of the main challenges faced by automated documentation tools is to capture the information needs of developers [70, 71, 92]. For example, Robillard et al. [82] discussed that research in automated documentation should produce results that better support the information needs of developers. Treude et al. [92] proposed an automated tool to extract tasks from documentation to help bridge the gap between the documentation structure and information needs of developers. Also, the increasing popularity of Q&A websites as a source of documentation has been explored by researchers to capture the information needs of developers [58, 97].

Nonetheless, most of the existing approaches to automate programming language documentation do not generate information that directly supports the decision-making process of designers, producers, and curators of documentation (i.e., documenters). Instead, most of the existing automated approaches focus on supporting program comprehension by developers (i.e., the users of the documentation) [62]. Although the automatic generation of documentation [68] and the identification of information needs of developers can be leveraged to support documentation activities, documenters have different requirements than developers that passively consume the documentation. For example, due to limited resources, documenters need to prioritize documentation effort [75]. As a consequence, automated techniques to support documenters' decision-making are necessary, particularly those techniques that are focused on external documenters that produce material about the *usage and features of a programming language*, without any particular association with a specific software system. Such documenters, as described in Section 2, are typically organized as dedicated teams around a specific programming language.

3.2 Topic Modelling Q&A Websites

Q&A websites (e.g., Stack Overflow) are popular venues where developers discuss technical aspects of programming. These websites allow users to post *questions* about a topic and to obtain *answers* from other users. A *post* in a Q&A website combines the question and the associated answers. Such information has been leveraged by software engineering researchers to understand what developers discuss about a variety of technologies. Information from posts is used to investigate the topic trends and challenges in areas such as mobile development [85], security [98], blockchain [94], machine learning [53], big data [52], deep learning frameworks [59, 66], configuration as code [80], concurrency [45], internet of things [48], chatbots [44], and new programming languages [57].

In prior works, the discussion topics in Q&A websites are learned via unsupervised topic models trained over the textual data of the posts [54]. Topic models learn two hidden components from the data of Q&A websites. The first component is the set of topics, with each topic being described as a probability distribution of word types. The second component is the set of documents, with each document being described as a probability distribution of topics (each document corresponds to a unique post). By associating topics with documents (posts), researchers leverage the metadata of the posts to calculate metrics that denote the popularity and the difficulty of the topics [44, 45, 47, 52] (e.g., the popularity of a topic is measured as the number of times a topic is associated with a post).

The usage of unsupervised topic modelling techniques incurs the cost of topic labelling. After using an unsupervised technique to extract the set of topics from the data, researchers must manually inspect the words with the largest probability to label a topic. However, it is not trivial to systematically assign labels to topics, and this task typically involves manually analyzing the top words in each topic, measuring the inter-rater agreement between researchers [57], merging related topics [52], and discarding meaningless topics [53]. Existing efforts to systematically assign labels to topics involve the derivation of reference architectures and the posterior mapping

of topics to layers of the architecture [66, 94]. Still, topics can be difficult to be interpreted, and some are especially hard to make sense of due to ambiguities. For example, Hindle et al. [67] show that domain experts are more likely to produce accurate topic labels than non-experts. Moreover, topic models are susceptible to over-representation of frequent (i.e., more general) topics—an effect called "rich topics get richer" [63]. As a result, unsupervised techniques hardly represent all the topics that agree with the researchers' knowledge about the studied domain, especially when topics of interest are underrepresented in a corpus. To address this challenge, one can adopt a semi-supervised topic model that uses encoded domain knowledge to bias the derivation of topics towards topics of interest.

3.3 Topic Modelling with Domain Knowledge

Anchored CorEx [64] is a semi-supervised topic modelling technique that does not assume a probabilistic generative model and avoids the need of adjusting parameters of a prior probability distribution such as in LDA-based topic models [55]. Anchored CorEx also accepts the integration of domain knowledge in the form of *anchor words*, i.e., a set of keywords that are associated with the topics of interest. These anchor words act as markers that help the model to distinguish particular topics from others.[5] Therefore, the generated topics can be pre-labelled according to the assigned meaning to their associated set of anchor words.

3.3.1 Anchored Correlation Explanation. The CorEx approach to topic modelling finds the set of topics that better "explain" the dependence (a.k.a. correlation) between each topic (a group of words) and the documents in the data [88]. Consequently, the learned parameters by CorEx are interpreted as the "association" between the topic and a document. Estimating dependence is a powerful characteristic of CorEx to represent topics without having to adjust any distribution parameter. The generated topic models by CorEx assign an individual weight for each topic in a document (i.e., the sum of the topic probabilities vary from 0 to m, where m is the number of topics). In the CoreEx theory, the *total correlation* is expressed as $TC(X_G; Y) = \sum_{i \in G} I(X_i : Y) - I(X_G : Y)$, where I is the mutual information of two random variables, X_i is a word type, X_G is a specifc subset of all word types G, and Y represents a topic. CorEx finds the configuration of topics that are the "most informative" of the words and documents by maximizing $TC(X_j; Y_1, \ldots, Y_m)$, where $Y_1, \ldots, Y_m$ are topics and X_j the corresponding set of word types of each topic, for $j = 1, \ldots, m$. Each topic contributes differently to the total correlation of the model. In turn, each document of the corpus contributes differently to the total correlation of a topic.

An extension to CorEx is Anchored CorEx [64], which accepts the incorporation of domain knowledge into the topic models. A set of anchor words [51] representing topics of interest is input to the model, with different anchoring strategies producing different results in terms of topic derivation. When anchoring for topic separability, the best strategy is to assign a set of (potentially exclusive) anchor words to each topic. During training, the association between anchor words and each topic is emphasized by a parameter β that controls the anchor strength. The ability to incorporate anchor words to nudge the topic's derivation is a valuable feature of CorEx, as it produces topics that capture a variety of related concepts to Rust and that are of interest to documenters.

In our study of the conceptual alignment of Rust documentation, the ability to cluster posts into topics is one of the most critical quality attributes of a topic model, since the studied metrics (described in Section 4.2) are based on the probability of a topic in a document. Regarding the ability to cluster documents into topics (by associating a document to the topic with the highest probability), Anchored CorEx is shown to be better or comparable to other semi-supervised topic models

[5] Anchor words is a related concept to prototypes in general clustering [72].

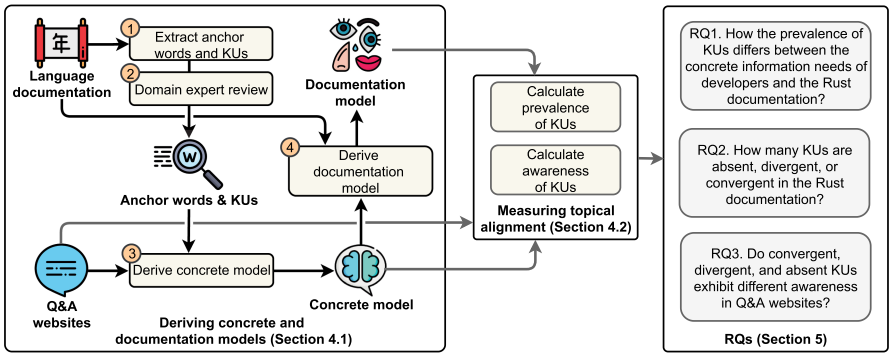


Fig. 1. The overview of our approach to studying the topical alignment of Rust documentation.

in two benchmark datasets (Disaster Relief Articles and 20 News Group) [64], with respect both to the intracluster quality (homogeneity) and the inter clusters quality (adjusted mutual information).

4 APPROACH

This section describes our approach to assess the topical alignment of the documentation of programming languages (Section 4.1) and to calculate metrics for our empirical study of the topical alignment of Rust documentation (Section 4.2). Figure 1 depicts the overview of our approach.

4.1 Deriving Concrete and Documentation Models

We adopt three main steps to derive the concrete and the documention models. We first extract anchor words and KUs from the language documentation (Section 4.1.1) and review the extraction results with a domain expert. In the next step, we derive a semi-supervised topic model from Q&A websites using the extracted anchor words to guide the topics derivation (Section 4.1.3). Afterwards, we derive the topics of the official language documentation by leveraging the topics of the concrete model that was built in the previous step (Section 4.1.4).

4.1.1 Extract Anchor Words and KUs. A KU is defined as a group of thematically related concepts within a general knowledge domain (e.g., a programming language). In the knowledge domain of programming languages, each KU encapsulates a set of related concepts that need to be understood by developers. For example, developers need to understand how to operate different *primitive data types* (e.g., scalars, arrays, and tuples) to properly use the language. Hence, primitive data types can be identified as a KU of a programming language. The documentation of a programming language describes KUs such that developers can understand and operationalize the different concepts of the language.

To identify KUs for our case study on Rust, we thoroughly read the official documentation of the language and categorize its contents into KUs. More specifically, we analyze the contents of eight documents [22] that cover the major relevant concepts of the language and are officially distributed with the standard Rust installation (in particular, we analyzed the documentation that is distributed with Rust version 1.49.0), namely, The Book [23], The Reference [24], The Cargo Book [13], The rustc Book [26], The rustdoc Book [27], Rust by Example [19], Nomicon [28], and the Edition Guide [16]. During the categorization of the Rust documentation into KUs, we also extract a set of associated keywords with each KU. Such keywords are used as the anchor words of our semi-supervised topic models.

We adopt an inductive procedure to extract and categorize anchor words from the Rust documentation, similar to the thematic synthesis procedure as described by Cruzes and Dyba [61]. Our

objective is to derive a domain knowledge model that represents the described concepts by the official Rust documentation and use this model as input to our semi-supervised topic model (see Section 3.3.1). We derived a domain knowledge model with three abstraction levels (from the more concrete to the more abstract): first, the set of anchor words; second, the associated KU with the set of anchor words; and third, the respective KU category. Our inductive procedure was carried out by the first author, who screened the contents of each chapter of the eight official Rust documents and identified segments of text that contain related keywords with the explained concepts. Each of the identified keywords is listed as a candidate anchor word that is iteratively refined during this inductive procedure. Each chapter's title, along with the associated sections and subsections, is used as a start point to determine the associated KU with the candidate anchor words. For example, The Book [23] has a chapter entitled "Generic Types, Traits, and Lifetimes," which explains different concepts related to the topics of generic, traits, and lifetime. Each of these three topics is assigned as a KU, while the related concepts that are explained in the chapter are assigned as associated anchor words with each KU. As we learn about the contents of the chapters, we refine the candidate anchor words and their associated KUs. At the end of this iterative process, we induced 375 anchor words grouped into 44 KUs. Finally, we induced four categories for the 44 KUs, which describe how the KUs are conceptually understood by developers that are learning Rust using the documentation.

After deriving anchor words and KUs from the eight aforementioned documents, we examine three Rust API references [20, 21, 25] in search of additional anchor words that complement the initial ones. However, we found that the contents of the three API references cover the same anchor words that were extracted from the other eight official documents. In addition, the three API references are mostly made of over-specific descriptions of concrete features of the language (e.g., the associated methods with 32-bit integers). We avoided associating such over-specific anchor words to our KUs (e.g., by avoiding associating concrete primitive types such as `i32` to our KU for primitive types KU) to prioritize the generalizability of the derived semi-supervised topic models from our domain model knowledge. Therefore, we did not add any anchor words from the analysis of the API references.

4.1.2 Domain Expert Review. We validate the extracted KUs and anchor words from the Rust documentation with a domain expert who has actively participated in the Rust documentation community[6] for several years. The domain expert suggested the creation of three additional KUs (and their respective anchor words) as well as the splitting of another KU and additional anchor words for some of the KUs that were originally derived. At the end of this process, we ended up with 47 KUs and their associated anchor words (the associated anchor words with each KU are described in Table A.1 of Appendix A). The KUs are grouped into four categories and shown in Table 1. The KUs are categorized according to how they are conceptualized and operationalized by developers who want to learn from the documentation. For example, the category of KUs for development tools represents as a set of concepts related with the tools that support the development of Rust applications. Similarly, the category of KUs for data types encompasses the data structures that are natively provided by Rust.

4.1.3 Derive Concrete Model. The concrete model of a programming language documentation represents the legitimate information needs of developers, as expressed by the voluntary discussions in Q&A websites. Figure 2 shows the performed steps to derive the concrete model for our Rust case study. We initially *collect and proccess data* from Q&A websites to generate our corpus.

[6] https://prev.rust-lang.org/en-US/community.html.

Table 1. Derived KUs from the Official Rust Documentation and Their Respective Categories

Category	Description	Knowledge Units
Data types	Data structures that are natively provided by the Rust language.	Advanced types, Collections, Enumeration, Generics, Primitive types, Smart pointers, and Structs
Development tooling	Tools to support the development of Rust applications.	Build, Compilation, Debugging, Dependency management, Documentation, IDE, Installation and setup, Modularization, Testing, and Linters
Language features	Functionalities and attributes that are natively provided the Rust language.	Aliasing, Advanced Functions and Closures, Casting, Control flow, Distribution channels, Exception handling, File I/O, Foreign Functional Interface, Functional language features, Lifetime, Macros, Mutability, Object orientation, Operators and symbols, Ownership, Patterns and matchings, Traits, and Unsafe Rust
Programming niche	Specialized set of features that suits particular applications of Rust.	Audio, Async, Command line arguments parsing, Command line interface, Concurrency, Database, Embedded development, Formatted print, Game development, Logging, Network development, and Web development

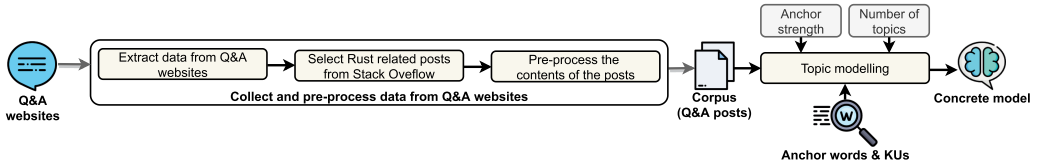


Fig. 2. The performed steps to derive the concrete model.

We then use Anchored CorEx [64] (see Section 3.3) to derive topics from the corpus, setting the anchor strength and number of topics parameters. We use the extracted anchor words of Section 4.1.1 as prototypes [65, 89] to each of the derived topics. In the following, we describe how we collect data from Q&A websites and how we derive our topic model.

Collect and Pre-process Data from Q&A Websites. Our data collection process encompasses three steps: *extract data from Q&A websites*, in which we obtain data from Stack Overflow—a popular Q&A website for general programming-related discussions—and Rust Forum—a specific Q&A website maintained by the Rust community, *select Rust-related posts of Stack Overflow*, in which we select Rust-related posts from Stack Overflow, and *pre-process the contents of the posts*, in which we prepare our data for our topic modelling.

Extract Data from Q&A Websites: We extract metadata about the posts of two Q&A websites: Stack Overflow and Rust Forum. Data from Stack Overflow is publicly available through the Stack Exchange Data Explorer [29] tool, from which we collect the `Posts.xml` file containing metadata about the posts (the file was downloaded on January 11th, 2021). Each post entry consists of either a question or one of its associated answers. We match the `Id` field of a question (i.e., an entry whose `PostIdType` field is equal to (1) with the `ParentId` field of an answer (i.e., an entry with `PostIdType` equal to (2) to link the question and the associated answers that belong to the same post. Table 2 describes the collected metadata of each post in Stack Overflow.

The Rust Forum website is built upon the open-source Discourse [14] platform, which allows data to be collected using an API [15]. We start by collecting all the questions[7] under the "help"

[7]We access the https://users.rust-lang.org/c/help/5.json?page=page_number API endpoint, where `page_number` is a placeholder to paginate the results.

Table 2. The Posts Metadata Collected from Stack Overflow and Rust Forum

Metadata	Description	Stack Overflow	Rust Forum
Post contents	The textual information of the question and answers.	The `Body` field of the question and the associated answers.	The `cooked` field of the question and the associated answers.
Post title	The associated title with the post.	The `Title` field of the question.	The `title` field of the question.
Accepted answer date	The date of the accepted answer of the post.	The `CreationDate` field of the answer whose `Id` field matches the `AcceptedAnswerId` field of the question.	The `created_at` field of the answer whose `accepted_answer` field is true.
Number of answers	The number of associated answers with the post.	The number of `ParentId` fields of the answers that match the `Id` field of the question.	The number of `topic_id` fields of the answers that match the `id` field of the question.
Tags	The user assigned tags to the post.	The `Tags` field.	–

category (i.e., the category of questions that are meant to request help from users of the forum). We select questions of the "help" category because this is the category under which developers discuss technical issues about the usage of Rust to solve programming problems and makes up more than 65% of all posts (the other two larger categories of posts include "uncategorized" and "announcements" that, together with "help," make up more than 92% of all posts). After obtaining metadata of all questions, we collect the metadata of all answers[8] associated with each question. Data from Rust Forum was collected on February 9th, 2021. Similar to the data from Stack Overflow, a post combines the original question and its associated answers. We use the `id` field of a question and the `topic_id` field of an answer to link the question and the associated answers of the same post. In total, we collected 14,520 questions from Rust Forum along with 81,225 answers. Table 2 describes the collected metadata of each post in the Rust Forum and on Stack Overflow.

Select Rust-related posts of Stack Overflow: In Stack Overflow, developers can discuss the solution of problems related with any programming language. Since we collect all posts from Stack Overflow, we need to select posts that are related to Rust. To this end, we leverage the user-assigned tags to identify all posts of interest. In particular, we search for all tags in Stack Overflow that have "rust" as a prefix, ending up with a total of 12 tags (including the "rust" tag). In addition, we derive a set of keywords from the language's documentation [22] that Stack Overflow users could potentially use to tag a Rust-related post (i.e., only keywords that effectively exist as a tag in Stack Overflow are considered). The derivation of those keywords is performed by the first author, who read the title of each section and subsection of the Rust Book [23] and used the Stack Overflow website to find whether the section titles are used as a tag. Table B.1 in Appendix B shows the initial set of 13 Rust-related tags that were are from this procedure. During the derivation of keywords from the documentation and the comparison of these keywords with the tags in Stack Overflow, we identified ambiguous keywords that could be used to tag posts related with other programming languages (e.g., "generics," "closure," and "enum"). For sanity checking, we manually analyze all tags of the posts that are tagged with any of the derived keywords, then we remove keywords that are associated with unrelated posts. For instance, many posts with the tag "generics" also have the tag "java" and, for this reason, we remove "generic" from our initial tag set. Table B.1 in Appendix B

[8]We access the https://users.rust-lang.org/t/-/topic_id.json?page=page_number API endpoint, where `topic_id` is a placeholder for the question `id` field and `page_number` is a placeholder to paginate the results.

shows our initial set of Rust-related tags that were effectively used to retrieve Rust-related posts from Stack Overflow.

After determining the initial set of keywords, we adopt a similar approach to prior works [74, 85, 94] to expand the initial set of tags and to remove irrelevant tags. More specifically, to expand the initial set of tags, we first search for all posts that contain any of our initial tags of Table B.1, from which we identify an additional 2,887 tags that are associated with those posts. To remove any irrelevant tags out of the 2,887 tags, we perform a filtering process based on two thresholds, namely, ***tag exclusivity threshold*** **(TET)** and ***tag significance threshold*** **(TST)**:

$$\text{TET}_{tag} = \frac{\text{Number of Rust posts}_{tag}}{\text{Number of total posts}_{tag}} > 50\%,$$

$$\text{TST}_{tag} = \frac{\text{Number of Rust posts}_{tag}}{\text{Number of Rust posts with the most popular tag}} > 1\%,$$

where:

$\text{Number of Rust posts}_{tag}$ = number of posts tagged with both tag and any of the initial Rust-related tags.
$\text{Number of total posts}_{tag}$ = number of posts tagged with tag.
The most popular tag = "rust," which tags 19,335 posts tagged with any of the initial Rust-related tags.

The rationale for filtering out tags with $\text{TET}_{tag} <= 50\%$ is to remove tags that are not exclusive of Rust-related posts. The rationale for filtering out tags with $\text{TST}_{tag} <= 1\%$ is that tags with high TET can be associated with a very low number of Rust-related posts. In these cases, TST will indicate a low confidence level in the tag's significance to denote a post of interest. We manually inspect the tags that survive our threshold criteria to select the tags that are exclusively related to Rust. This procedure consists of reading a list of eight tags and selecting those that are related to Rust and different from the pre-selected tags shown in Table B.1. A total of three other tags are added to our tag set during this manual inspection process: "lifetime," "serde," and "borrowing." By selecting the posts with any of these tags, we ended up with 19,603 questions along with 22,715 answers (out of 54,242,283 questions and 82,442,747 answers from Stack Overflow).

Pre-process the contents of the posts: To prepare our data for our topic modelling, we concatenate the post title and the post contents (see Table 2) of each Q&A website. We then combine the posts of both Q&A websites in a unique set of posts. Finally, we perform six pre-processing steps:

(1) Remove code snippets within `<code></code>`.
(2) Remove HTML tags.
(3) Remove punctuation and non-alphabetic characters.
(4) Replace compound anchor words with their underscored format.
(5) Lemmatize.
(6) Remove stop words.

To remove HTML tags, we use the BeautifulSoup[9] library. We replace any occurrence of a compound anchor word in a post by its underscored format. For example, whenever we identify the occurrence of the "primitive type" anchor word in a post, we replace such an occurrence with "primitive_type." The rationale for this preprocessing step is to ensure that compound anchor words can be properly captured during the derivation of topics. For lemmatization and stop word

[9]https://pypi.org/project/beautifulsoup4/.

removal, we used the NLTK[10] library. After performing the six aforementioned pre-process steps, we tokenize the contents of the posts and obtain our corpus for the Q&A websites (see Figure 2).

Topic Modelling. We use the Anchored CorEx implementation [12] to automatically derive topics from the contents of the Rust-related posts and to assign a set of topics to each post. Two parameters are given to the topic modelling technique: the number of topics and the anchor strength (i.e., the probability weight assigned to the anchor words in relation to the other words in a topic).

Number of topics: Ideally, our topic models should have the same number of topics as the number of derived KUs, since this characteristic ensures a one-on-one mapping of anchor words and topics, which facilitates our analyses in terms of KUs. Note that such one-on-one mapping refers to the correspondence between anchor words and semi-supervised topics, with the associated KU with the anchor words serving as a label to the topic. As we adopt Anchored CorEx to derive topics from the Rust-related posts, our model can denote the occurrence of multiple topics in a single post, or the occurrence of no topics at all if the textual content of the post does not correlate with any of the extracted topics from the rest of the posts in the dataset. We perform three steps to determine whether we can obtain quality topics in a model that has the same number of topics as KUs (or whether, instead, more topics would be needed). In our first step, we derive two tentative topic models, one with 47 anchored topics (one topic per KU) and another model with the same 47 anchored topics and an additional non-anchored topic. We then manually analyze the additional non-anchored topic to verify the interpretability of this topic. We find that this additional topic is composed of commonly used words by developers in Q&A websites that cannot be directly related to a KU of Rust. Specifically, the top-five words in this topic are think, make, thing, case, and need. In a second step, we measure the difference between the coherence (a well-known metric for topics quality) [83] of the associated topics with each of the two tentative models. Specifically, we determine whether the increase of the number of topics from 37 to 38 topics translates into a significant increase in coherence. We found that the difference between the u_mass coherence of the two tentative models is only 0.090, which does not justify the preference for any specific model based on the coherence of their topics. Finally, we decompose the models' overall **total correlation (TC)** (see Section 3.3) into a distribution of individual topic TCs that are calculated during model training [88]. If we observe relatively small values of topic TC in this distribution (in relation to the higher values of TC), then the number of topics of the model is sufficient to explain the correlation in the data, and additional topics will not contribute to the overall TC [64]. Figure 3 shows that the model with 47 anchored topics presents a good compromise between the ability to explain the correlation in the data without additional topics and the requirement of one-on-one mapping of anchor words and topics.

Different from generative topic models such as LDA, where the topic probability distribution for a document sums up to one, Anchored CorEx assigns an individual probability for each topic in a document. This characteristic implies that a document might have no topics associated with it, especially when the anchor words of the topics are not representative (e.g., they are insufficient). After deriving the concrete model, we did not observe any documents without an assigned topic, increasing our confidence that the 47 selected topics are suitable to represent the word associations in the Q&A posts. Assuming that a one-on-one mapping exists between anchor words and topics, we label a topic with the KU that is associated with the set of anchor words of that topic. For this reason, during our analysis of the topic model (Section 5), we use the terms KU and topic interchangeably. In our formal definitions, we denote the set of topics as K.

[10] https://www.nltk.org/.

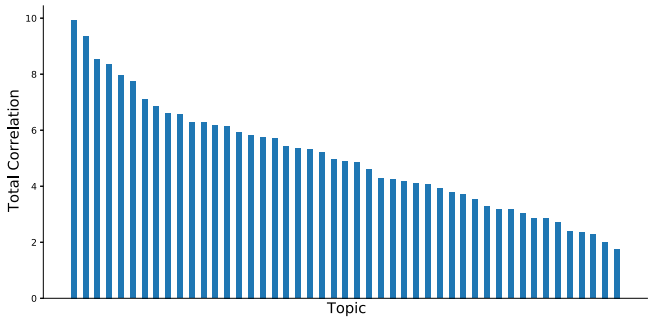


Fig. 3. The distribution of the topic TC of the model with 47 anchored topics.

Anchor strength: To determine the anchor strength (see Section 3.3), we tentatively build models with anchor strength parameters $\beta = \{2, 5, 10, 15\}$. For each of the tentative models, we manually inspect the generated topics and qualitatively evaluate how well the top 20 words in the topic agree with the associated KU with that topic (via assigned anchor words). More specifically, we manually analyze the derived topics to ensure that there is a correspondence between the associated anchor words and the top words of the topics. To complement our sanity check, we also manually analyze several random instances of the Q&A posts to validate the correspondence between their textual contents and the associated word types of the derived topics. We observe that the textual contents of the Q&A websites (question and associated answers) have a strong relationship with the top words in the generated topic in the sense that, in general, the words used in the Q&A posts either appear in or are correlated with the topics of the post. We illustrate the findings of this latter sanity check with the following three category of examples: (1) Some Q&A posts denote information needs that are specifically associated with an assigned topic. For instance, the Q&A post entitled "*How to import all macros, derives, and procedural macros in Rust 2018 without using extern crate?*" [38], which was assigned with the macros topic, is a specific inquiry about the usage of macros in Rust. Another example is the Q&A post entitled "*Log4rs - how to use different levels for appenders*" [41], which was assigned with the logging topic (among other topics as well) and is a specific question about the usage of a log-related library in Rust; (2) Some Q&A posts denote information needs that are indirectly related to an assigned topic, with top words associated with the topic being found in the post. For instance, the Q&A post entitled "*Is there an idiomatic way to implement the component pattern?*" [40] was assigned with the game development topic and inquires about how "*to implement the Component pattern commonly used in games*" in Rust. Another example is the Q&A posts entitled "*Intercepting libc readlink with a RUST LD_PRELOAD program hangs when applying to cargo build*" [39], which is assigned with the build topic and inquires about the usage of the linker preload mechanism to trace the cargo build command execution; (3) Some Q&A posts denote information needs that are indirectly related to an assigned topic, without top words associated with the topic being found in the post, but instead inferred by our underlying topic model. For instance, the Q&A post entitled "*How can I convert it back into* `bson::document::Document` *on the server side?*" [37] was assigned with the database topic. Although the post does not contain any database-related keyword, it was correctly associated with this topic due to correlation between the words in the post and the associated words with the database topic.

We also observe that the produced topics with different anchor strengths are very similar and, in practice, the adoption of a semi-supervised topic modelling to derive the concrete model is able to generate topics that agree with the KU, regardless of the adopted anchor strength. We can

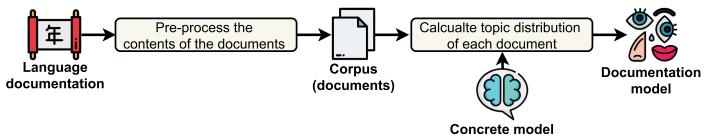


Fig. 4. The performed steps to derive the documentation model.

verify that many additional words beyond the anchor words are also correctly associated with the top 20 words of the topics and that such additional words are related to the represented domain knowledge by the KU. During our manual analysis, we observe that topics that occur in a smaller number of documents are at risk of having spurious words associated with them (which occur with any topic model). Therefore, we focus on such topics and strive to select the tentative model with the least number of spurious words in the top 20 words of these topics. Based on this qualitative evaluation, we conclude that an anchor strength parameter $\beta = 10$ produces the best topic-word distribution among the tentative models. The 47 derived topics are shown in Table C.1 of Appendix C. Deriving the concrete model results in a topic model M_{Concrete} with estimate probabilities $p(y|x)$ of a certain topic y to occur in a document (i.e., a Q&A post) x, given the words in x.

4.1.4 Derive Documentation Model. The derivation of the documentation model encompasses the two depicted steps of Figure 4. First, we *pre-process the contents of each document* of the official Rust documentation, from which we obtain a corpus of the documentation. Next, we *calculate the distribution of topics that occur in each document* of the corpus.

Pre-process the Contents of the Documents. The official Rust documentation consists of a series of books. In turn, each book consists of a set of chapters that cover different aspects of the language. To build the corpus of the official language documentation, we collect each chapter of the 8 documents described in Section 4.1.1, in addition to the chapters of three reference API documents, namely, The Rust Core Allocation and Collections Library [20], The Rust Core Library [21], and the Rust Standard Library [25]. These 11 books represent the complete documentation distributed with the standard installation of Rust version 1.49.0. The books are distributed in a standard `html` format that is automatically generated by the mdBook tool.[11] To pre-process the contents of each book chapter, we initially parse the respective `html` files and extract the text within any `<p></p>` tags. We then concatenate all the extracted text that is associated with the same `html` file (as each file contains the material of one book chapter). Finally, we apply the same preprocessing steps described in Section 4.1.3 and tokenize the extracted text from the Rust documentation. In total, our corpus contains 4,754 documents, each one corresponding to a book chapter.[12] We then use the same topics (word type probability distribution) of the concrete model to build our documentation model.

Calculate the Topic Distribution of Each Document. The documentation model of a programming language represents the topics that occur in each document of the language's documentation. By deriving the documentation model, our objective is to measure the topical alignment of Rust documentation by comparing the prevalence of the topics of the concrete model against the prevalence of the topics of the documentation model (further details about the calculation of topic prevalence are given in Section 4.2). The comparison between topics of the concrete and the documentation model is based on two assumptions. Our first assumption is that we only compare topics

[11] https://github.com/rust-lang/mdBook.

[12] In total, there are 4,771 chapters in the books of the official Rust documentation. However, 17 of such chapters do not contain any textual information within `<p></p>` tags, leaving us with a total of 4,754 useful chapters.

that occur in both models. As we derive the concrete model using induced anchor words from the language's documentation (i.e., the KUs of interest that documenters want to measure topical alignment), topics that occur in the concrete model also occur in the documentation by definition. Note that we make no assumption on the relationship between the KUs of interest for documenters and the subjects that are effectively discussed by developers in the Q&A posts. Therefore, topics that co-exist in the concrete and the documentation models can represent a proper subset of all the discussed subjects in Q&A posts. Hence, to validate our assumption, we need to suppress topics that occur exclusively in the documentation model. The suppression of exclusive topics of the documentation model does not limit the measurement of topical alignment, as we are not interested in making claims regarding topics that occur only in the documentation. For instance, we do not suggest removing topics that are not mentioned in any Q&A post from the documentation, as a missing topic from Q&A websites might be due to the wide coverage of that topic by the documentation. Our second assumption is that we perform a paired comparison of topics between the Q&A posts and the official documentation. By paired comparison, we mean comparing the same topics (word types distribution) between the concrete and the documentation models.

To satisfy our two assumptions, we leverage the M_{Concrete} model to calculate the probability of each topic from M_{Concrete} to occur in each document of the official documentation collection. This procedure contrasts with the derivation of a new semi-supervised topic model from data of the official language's documentation, potentially containing different topics than the topics of the concrete model. To derive the documentation model, we apply the same optimization procedure as described in Section 3.3 to assign topics to each document of the language's official documentation. More specifically, for each document of the official documentation, we calculate the configuration of topics Y_{Concrete} from the concrete model that maximizes $TC(X_{\text{Documentation}}; Y_{\text{Concrete}})$, where $X_{\text{Documentation}} \subseteq X_{\text{Concrete}}$ (i.e., the word types of the documentation model are constrained to the word types of the concrete model). Note that each topic in the derived documentation model has the same associated KU with each of the respective topics of Y_{Concrete}. We use the `predict` function provided by CorEx [12] as the implementation of our optimization procedure. The output of this procedure is the probability of each topic y of Y_{Concrete} in each document x of the official documentation.

4.2 Measuring Topical Alignment

The concrete and documentation models represent an estimate of the probability distribution $p(y|x)$ of a certain topic y in a post or a document x, given the word types in x. We leverage the estimated probabilities of each model to calculate different metrics that measure the topical alignment between the concrete and the documentation models. As we have a one-on-one mapping between KUs and topics, our metrics denote specific characteristics of a KU (e.g., the frequency of a KU) from the measure of properties of the associated topic (e.g., the frequency that the associated topic with the KU occurs in the documents of the corpus). Figure 5 shows the taxonomy of our KU metrics. Two higher-level groups of metrics are the *prevalence* metrics (studied in Sections 5.1 and 5.2) and the *awareness* metrics (studied in Section 5.3). Three of the prevalence metrics, namely, *occurrence*, *co-occurrence*, and *dominance*, are used as the basis for the definition of other four metrics: *frequency*, *co-frequency*, *coverage*, and *affinity*. Three additional metrics are defined as an awareness metric: *attraction*, which measures the response rate associated with a topic in the Q&A websites, *attention*, which measures the number of responses to a post in a topic, and *agreement*, which measures the rate of accepted answers associated with a topic. The awareness metrics are calculated from metadata of the Q&A websites (see Table 2) and, therefore, are defined only for the concrete model (as we do not have a direct association between a Q&A post and a document from Rust documentation).

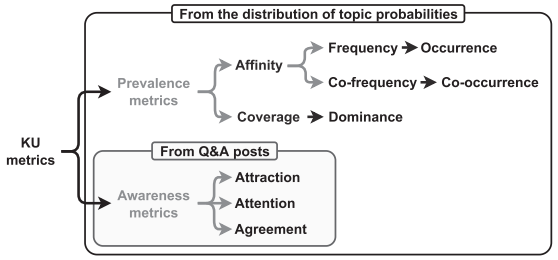


Fig. 5. The taxonomy of our KU metrics.

4.2.1 *Calculate Prevalence of KUs.*

Occurrence: Q&A posts and documents can involve related concepts to different KUs. Therefore, we wish to denote the occurrence of a topic in a post or document. A topic y occurs in a post or document x if the topic probability is larger than 50%. The choice of a threshold of 50% was motivated by the similarity between our models' estimated probabilities and the usage of sigmoid functions in binary logistic regression, for which the adoption of a 50% threshold is a widespread practice:

$$occurrence(y, x) = \begin{cases} 1, & \text{if } p(y|x) > 0.5 \\ 0, & \text{otherwise.} \end{cases}$$

Co-occurrence: More than one KU might occur in the same document. Hence, we wish to denote the co-occurrence of two topics in a document. Two topics y' and y'' are co-occurrent if both topics occur in a document x:

$$co\text{-}occurrence(y', y'', x) = \begin{cases} 1, & \text{if } occurrence(y', x) + occurrence(y'', x) = 2 \\ 0, & \text{otherwise.} \end{cases}$$

Dominance: Although multiple KUs might occur in the same post or document, each of the KUs in a post or document has a different ability to denote the main subject topic. To differentiate a KU that is the main subject of post or document from a KU that is as a supplementary subject, we wish to denote the most representative (a.k.a. dominant) topic from the set of topics in a post or document. The dominant topic of a post or document x is the topic y with the largest probability:

$$dominance(x) = y : p(y|x) = \text{argmax}(p(y|x)), \forall x \in P.$$

Frequency: A KU can occur more or less frequently in the posts and documents. Hence, we want to measure how often the same topic occurs across the posts or documents. The frequency of a topic y is measured as the number of posts or documents x in which y occurs:

$$frequency(y) = \sum_{x \in P} occurrence(y, x).$$

Co-frequency: Two KUs can occur in conjunction in the same post or document. Hence, we want to measure how often each pair of KUs occurs in a post or document. The co-frequency of a pair of topics y' and y'' is the total number of times that these two topics occur in the same post or document x.

$$co\text{-}frequency(y', y'') = \sum_{x \in P} co\text{-}ocurrence(y', y'', x)$$

Popularity: The frequency with which a KU is the main subject topic of a post or document differs depending on the KU. Therefore, we want to measure how often a certain topic is the dominant topic of a post or document. The *popucoveragelarity* of a topic y is measured as the number of posts or documents x in which y is dominant:

$$coverage(y) = |\{x \in P : dominance(x) = y\}|.$$

Affinity: KUs that often co-occur in the same post or document share a high level of affinity. Although the relatedness of two KUs can be observed from the co-frequency metric, KUs that occur too infrequently in comparison with other KUs are penalized by this metric. We want to measure the *affinity* of a pair of topics normalized by the number of occurrences of both KUs. Affinity is calculated as the number of times that a pair of topics y' and y'' co-occur in a post or document divided by the mean frequency of the topics:

$$affinity(y', y'') = \frac{cofrequency(y', y'')}{(frequency(y') + frequency(y''))/2}.$$

The affinity metric represents the expectation of two KUs to occur in the same post or document. The metric values range from 0 to 1, with the minimum value representing that the two topics never co-occur and the maximum value representing that whenever one topic occurs in a post or document, the other topic also occurs.

4.2.2 *Calculate Awareness of KUs.*

Attraction: Associated posts with specific KUs can attract more attention from developers than those associated with others KUs. Q&A posts that do not receive any comment from developers indicate that developers are not seeking for related information about the KUs that occur in that post. Therefore, to denote the attraction of developers by a KU, we measure the *response rate of a KU* that is dominant in the assciated discussion with a post. Formally, let $a(x)$ be the number of answers of a post x. The *attraction* of a topic y is defined by:

$$attraction(y) = \frac{|\{x : dominance(x) = y \wedge a(x) > 0, \forall x \in P\}|}{coverage(y)}.$$

The attraction metric ranges from 0 to 1, with the minimum value occurring when none of the Q&A posts x for which the topic y is dominant have an answer. The maximum value occurs when all posts x for which the topic y is dominant have an answer. The dominant topic is used in this metric because we assume that posts without an answer will be better represented by the dominant topic. If the topic y is never dominant, then attraction is not defined for y.

Attention: The number of answers that a Q&A post receives can be understood as the amount of attention that developers draw in that post. Since multiple topics can occur in the same Q&A post, each topic can contribute with a different weight for attracting attention to that post. The *attention* of a topic y is the average probability $p(y|x)$ weighted by the number of answers in a post x:

$$attention(y) = \frac{\sum_{x \in P} \{p(y|x) \times a(x) : occurrence(y, x) > 0\}}{\sum_{x \in P} \{p(y|x) : occurrence(y, x) > 0\}}.$$

The attention metric represents how much a KU contributes to the total number of answers of the posts for which the KU occurs.

Agreement: The degree with which developers agree with the description of the solution of a programming problem can vary. Q&A posts that have an accepted answer indicate that developers agree with a certain solution to a problem. Therefore, we want to denote the association of a KU

with the developers' ability to agree with an accepted answer. Formally, let $a'(x) \neq 0$ if and only if the post x has an accepted answer. We define the *agreement* of a topic y as the proportion of posts x in which y occurs and that have an accepted answer:

$$agreement(y) = \frac{|\{x : occurrence(y, x) > 0 \wedge a'(x) \neq 0, \forall x \in P\}|}{|\{x : occurrence(y, x) > 0, \forall x \in P\}|}.$$

The agreement metric ranges from 0 to 1, with the minimum value representing that none of the posts that topic y occurs have an accepted answer and the maximum value representing that all posts that topic y occurs have an accepted answer.

5 RESEARCH QUESTIONS

This section describes the motivation, approach, and results of each of our RQs. Sections 5.1, 5.2, and 5.3 discuss RQ1, RQ2, and RQ3, respectively.

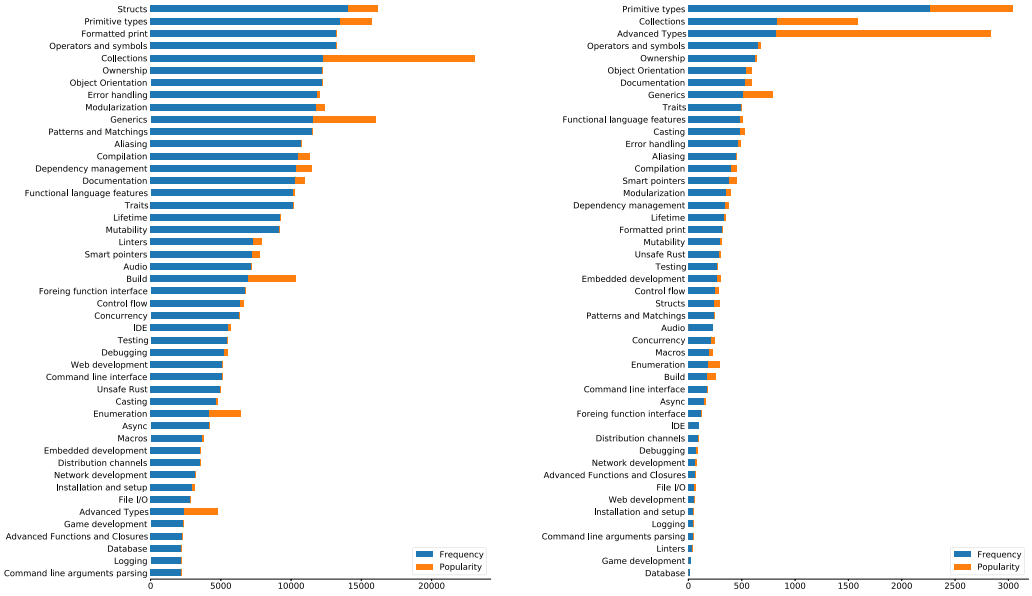
5.1 How the Prevalence of KUs Differs between the Concrete Information Needs of Developers and the Rust Documentation?

Motivation: Documenters need to assess how likely is the actual state of the documentation to match the information needs of developers and to reason about the timing for revising the contents of the documentation. In this RQ, we *quantify* the topical alignment between the concrete and the documentation model of Rust by determining how similar is the prevalence of KUs between these two models.

5.1.1 *Analysis of Frequency and Coverage Metrics.*

Approach: For both the concrete and documentation models, we analyze the distribution of the *frequency* and *coverage* metrics along the set of all KUs (see Section 4.2.1 for a definition of such metrics). With this analysis, we identify the most frequent and popular KUs in both models and observe the shape of such distributions. To measure the extent to which the frequency and popularity deviate from a normal distribution, we calculate the adjusted Fisher-Pearson standardized moment G_1 coefficient of skewness and interpret the absolute value of the coefficient as follows [96]: *almost symmetric*, if $-0.5 \leq G_1 \leq 0.5$, *slight skew*, if $0.5 < G_1 \leq 1.0$, or *skew*, if $G_1 > 1.0$. We then analyze whether there is a significant correlation between the coverage or the frequency of KUs between the concrete and the documentation models and whether a significant correlation exists between the two metrics within the same model. From this analysis, we observe the extent to which KUs are equally prevalent in the two models and whether KUs that are supplementary (as measured by its frequency) are also the main subject (as measured by its popularity) of the individual posts or documents. To measure the correlation, we calculate the Spearman ρ coefficient and interpret its absolute value as follows [81]: *negligible*, if $0 \leq |\rho| \leq 0.10$, *weak*, if $0.10 < |\rho| \leq 0.39$, *moderate*, if $0.39 < |\rho| \leq 0.69$, *strong*, if $0.69 < |\rho| \leq 0.89$, or *very strong*, if $0.89 < |\rho| \leq 1$. All hypotheses testing are two-sided and performed under a significance level of $\alpha = 0.05$.

Results: In terms of the rank of frequency and coverage of KUs, there is a strong agreement between the concrete and the documentation models. Figures 6(a) and 6(b) show both the frequency and coverage of each KU in the concrete and the documentation models, respectively. The frequency and the coverage of the KUs follow a skewed distribution (see Table 3), except for the distribution of the frequency in the documentation model. This observation indicates that few KUs account for most of the individual occurrences and dominances in the documents. Table 4 shows that there is a strong correlation between the frequency of the KUs of the concrete and the



(a) Concrete model (b) Documentation model

Fig. 6. The frequency and popularity of each KU (sorted by frequency) in the concrete and documentation models of Rust.

Table 3. The Skewness of the Distribution of the Prevalence Metrics in the Concrete and Conceptual Models of Rust Documentation

Model	**Metric**	**Skewness**	
		G_1 **coefficient**	**Significant?**
Concrete	Frequency	0.21 (almost symmetric)	Yes
	Coverage	4.36 (skew)	Yes
	Co-frequency	1.68 (skew)	Yes
	Affinity	1.46 (skew)	Yes
Documentation	Frequency	3.64 (skew)	Yes
	Coverage	4.99 (skew)	Yes
	Co-frequency	2.78 (skew)	Yes
	Affinity	1.23 (skew)	Yes

documentation models ($\rho = 0.72$, p-value $= 1.59 \cdot 10^{-8}$), as well as between the KUs coverage ($\rho = 0.73$, p-value $= 6.13 \cdot 10^{-9}$). In the concrete model, the frequency of the KUs weakly correlates with the coverage ($\rho = 0.53$, p-value $= 1.11 \times 10^{-4}$), suggesting the degree to which the same KU occurs either as a supplementary or as the main subject of the posts. In turn, the same correlation is moderate in the documentation model ($\rho = 0.65$, p-value $= 7.70 \times 10^{-7}$).

5.1.2 Analysis of the Co-frequency and Affinity Metrics.

Approach: We first analyze the distribution of the number of KUs that occur in the same post or document of each model. The objective of this analysis is two-fold: First, we want to analyze the extent to which more than one KU is associated with the same post or document, rendering

Table 4. The Correlation between the Same Metric for Two Different Models of Rust Documentation and the Correlation between Different Metrics for the Same Model of Rust Documentation

Metrics	Comparison	Constant	Correlation	
			ρ coefficient	Significant?
Frequency & Coverage	Different models, constant metric	Frequency	0.72 (strong)	Yes
		Popularity	0.73 (strong)	Yes
	Constant model, different metrics	Concrete model	0.36 (weak)	Yes
		Documentation model	0.65 (moderate)	Yes
Co-frequency & Affinity	Different models, constant metric	Co-frequency	0.79 (strong)	Yes
		Affinity	0.69 (strong)	Yes
	Constant model, different metrics	Concrete model	0.93 (very strong)	Yes
		Documentation model	0.75 (strong)	Yes

the co-frequency and affinity metrics valid. Second, we want to evaluate how the concrete model agrees with the documentation model in terms of the number of KUs per post or document. We use the one-sided Mann-Whitney procedure to test the null hypothesis that the number of KUs per post of the concrete model is significantly larger than the number of KUs per document of the documentation model ($\alpha = 0.05$).[13] In case the null hypothesis is rejected, we calculate the Cliff's Delta d coeficient [60] to measure the effect size of the difference. We interpret the coefficient according to the following thresholds [84]: *negligible*, if $0 \leq |d| < 0.147$, *small*, if $0.147 \leq |d| < 0.330$, *medium*, if $0.330 \leq |d| < 0.474$, or *large*, if $0.474 \leq |d| \leq 1$. As for the co-frequency and the affinity metrics, we first analyze the skewness of the distributions of the two metrics in the two models using the Fisher-Pearson standardized moment G_1 coefficient. This analysis shows whether there are pairs of KUs that are more prevalent than others. Then, for both the co-frequency and the affinity metrics of each pair of KUs, we analyze the Spearman ρ rank correlation between the concrete and documentation models, as well as within the same models. The coefficients of skewness and correlation are interpreted according to the thresholds given in Section 5.1.1.

Results: Multiple KUs occur in association within the same post or document. Still, compared with the documentation model, the concrete model tends to have a significantly larger number of KUs per post. Figure 7 shows that the median number of KUs that occur per post is 9 in the concrete model, while the median number of KUs per document in the documentation model is 2. The difference in the number of KUs per post or document between the two models is statistically significant (p-value < 0.05) and the effect size is large ($|d| = 0.748$). As for the co-frequency and affinity metrics, Table 3 shows that the distribution of both metrics are skewed in both models, suggesting that few pairs of KUs are responsible for most of the co-occurrences in the posts and documents (which also holds valid when we consider the co-occurrences relative to the average frequency of the two KUs). In addition, Table 4 shows that **there is a strong correlation between the two models with respect to the rank of the co-frequency and the affinity metrics.** The correlations between the co-frequency and the affinity metrics within the same model are very strong and strong, respectively, showing that the two metrics share similar factors that influence their values within a model.

[13]The null hypothesis was stated *after* visualizing the distributions of the number of KUs per document in the concrete and documentation models.

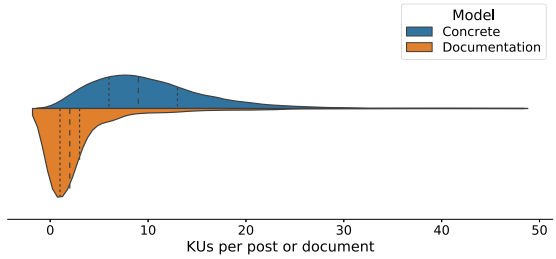


Fig. 7. The distribution of the number of KUs that occur in the same document for the concrete and the documentation models of Rust.

How the prevalence of KUs differs between the concrete information needs of developers and the Rust documentation?

There is a strong agreement between the concrete and documentation models of Rust in terms of the rank of frequency and coverage metrics. The agreement is also strong for the rank of co-frequency and affinity metrics. Although the co-frequency and affinity metrics have a high correlation within the concrete and documentation models, the same does not hold for the frequency and coverage metrics, for which the correlation within the models are weak and moderate, respectively.

5.2 How Many KUs Are Absent, Divergent, or Convergent in the Rust Documentation?

Motivation: Documenters need to determine which KUs differ between the concrete and documentation models. In particular, documenters can differentiate between KUs that are associated with three different *types of KU alignment*, based on the rank of KUs in the concrete and documentation models with respect to the frequency metric. The three categories of KU alignment are absent, divergent, and convergent. An *absent KU* is a KU that is low ranked in both models. Documenters should be aware of absent KUs, as they represent concepts that are rarely documented in both models. A *divergent KU* is a KU that is high ranked in the concrete model but is low ranked in the documentation model. Documenters should consider improving the amount (or enhancing the quality) of information about divergent KUs in the official documentation, as these KUs indicate that there is high demand of information by developers and a scarcity of the same information type in the official documentation. Finally, a *convergent KU* is a KU that is high ranked in both the concrete and the documentation models. Documenters should be mindful of convergent KUs to avoid allocating resources to document redundant information.

Approach: To categorize the KUs into convergent, divergent, and absent, we first divide the rank of KUs into three segments (in terms of the frequency metric), where the cutoff point of the segments are determined according to the percentiles of the frequency metric. We call a KU as *high ranked KU* if this KU is ranked above the associated cutoff point with the highest ranked segment. Similarly, we call a KU as *low ranked KU* if the KU is ranked below the cutoff point for the lowest ranked segment. As the distribution of the frequency metric has a different shape in the two models (see Section 5.1.1), we adopt different cutoff points depending on the model. We tried several combinations of cutoffs and we found that the configuration shown in Table 5 yields interpretable and informative results. To aid the interpretability of our analysis of the types of KU alignment, we present the findings using an infographics in which the line color represents the

type of KU alignment, while the line thickness represents the rank distance between the same KU on the concrete and documentation models.

The established cutoff points in our Rust case study are used to help us focus on KUs that potentially require topical alignment improvement and are relatively positioned in a different frequency rank. Our objective of using the cutoff points is to avoid arbitrarily pointing to specific KUs that require topical alignment improvement, without having a reproducible form of identifying convergent, divergent, and absent KUs. However, changing the cutoff points shown in Table 5 makes the classification of KUs more or less conservative. Depending on how a documenter wants to identify KUs whose topical alignment requires improvement, the cutoff points can be adjusted (or even other classification methods can be used). Hence, the cutoff points used in our case study might not generalize to other scenarios where the documentation has a different topical alignment. For example, if the low rank cutoff for the concrete and documentation models are always above a certain percentile (i.e., both low rank cutoffs are larger than zero), then there will always be some KUs that are deemed as absent. Documenters that want to be more conservative and identify fewer KUs as divergent or absent can decrease the low rank cutoff points or increase the high rank cutoff points. Conversely, documenters that want to be less conservative and identify more KUs as divergent or absent can increase the low rank cutoff and decrease the high rank one. Finally, documenters should consider not only the frequency rank of the KUs, but also the absolute frequency values to determine the cutoff points and to classify KUs into convergent, divergent, and absent. In addition, documenters should also rely on their knowledge about the documentation state to make an informed and critical decision of which KUs to pay attention to the topical misalignment. To ensure that no misleading results are reported in our article (i.e, absent and divergent KUs that do not require topical alignment improvement), we perform a validation step based using a qualitative analysis of developers' perception about the documentation state related to the identified KUs as absent and divergent.

To determine the extent to which developers perceive that the identified absent and divergent KUs by our approach in fact require better topical alignment, we perform a qualitative analysis on three external information sources. Our objective is to validate our results by identifying pieces of evidence that support developers' perceptions about the state of the documentation for absent and divergent KUs. All the analyzed information sources have the potential to contain voluntary statements from developers regarding KUs whose documentation is misaligned with their needs. Therefore, we analyze these three information sources in search of supporting evidence that the associated documentation with divergent and absent KUs requires better alignment with developers' information needs. Below, we describe each of the analyzed information sources:

(1) **Documentation-related issue reports (IRs) and pull requests (PRs):** We manually analyze the IRs and PRs that were created on the repositories where the official Rust documentation is maintained. Our objective is to identify which specific parts of the documentation require improvement.
(2) **Rust-related surveys:** We analyze the results of the existing Rust-related surveys to identify whether respondents have pointed out that the documentation of some KUs needs improvement.
(3) **Documentation-related Q&A posts:** We manually analyze Q&A posts to identify statements about the lack of alignment between developers' information needs and the documentation.

We search for evidence of topical misalignment for each of the divergent and absent KUs. For documentation-related IRs and PRs, we screen all information that was created after the release date of Rust 1.49.0 (December 31st, 2020) up to March 15, 2022 (the date of our analysis). The

Table 5. Adopted Cuttoff Points to Determine the High and Low Ranked KUs

Model	High rank cutoff	Low rank cutoff
Concrete	50th percentile	25th percentile
Documentation	75th percentile	50th percentile

reason for analyzing information within this specific time frame is to reduce the number of IRs and PRs that do not refer to the same documentation state as used to derive the documentation model. In total, we analyzed 1,442 IRs and PRs. We observe that the IRs and PRs are created for a variety of reasons, ranging from questions about the examples contained in the documentation, bug reports (e.g., a broken example), requests for specific translations, general discussions, and improvement suggestions. Therefore, during our qualitative analysis, we select (a) IRs that suggest documentation improvements and that are deemed as relevant (e.g., documenters explicitly agree with the relevance of the improvement suggestion) and (b) PRs that are merged into the documentation, while typos and other minor fixes are ignored. We also search for existing Rust-related surveys on the divergent and absent KUs. We primarily focus on surveys in which participants opine about the state of the documentation on the related KU. To the best of our knowledge, the only existing survey with this characteristic is related to the KU for game development. Finally, for documentation-related Q&A posts, we search for posts on the official Rust Q&A website containing the terms "doc," "docs," "document," "documents," or "documentation." We found 346 posts containing at least one of those keywords. We then identify the posts where developers explicitly mention the documentation of absent or divergent KUs.

During the qualitative analysis of the information sources, we infer which KUs are related to the analyzed information by observing different textual cues. For example, when analyzing Q&A posts and IRs, we observe references to specific domains of the language. Similarly, when analyzing PRs, we observe which sections of the documentation are referred to and the relation of those sections with the KUs. Finally, during our analysis, we also extract quotes that support the validation of our results and exemplify developers' information needs. We extracted a total of 63 quotes from the three information sources. Finally, we combine the supporting evidence gathered from the three different information sources to confirm the absent and divergent KUs that developers consider as requiring better topical alignment.

Results: We observe the existence of 10 absent KUs for macros, distribution channels, network development, installation and setup, file I/O, advanced types, game development, advanced functions and closures, database, logging, and command line arguments parsing. In addition to the absent KUs, **we observed the existence of 6 divergent KUs** for structs, patterns and matchings, linters, audio, build, and foreign function interface. Finally, **we also observe the existence of 10 convergent KUs** for primitive types, operators and symbols, collections, ownership, object orientation, error handling, generics, documentation, functional language features, and documentation. Figure 8 shows the absent, divergent, and convergent KUs in terms of frequency.[14]

Table 6(a) shows the proportion of KUs that pertain to each category grouped by the type of KU alignment (see Table 1 in Section 4.1 for a description of the category of KUs). We observe that 50% of the absent KUs are categorized as a KU for programming niche, suggesting that documenters should consider improving the documentation of these KUs to support the adoption of Rust for those niches. In addition, 40% of the absent KUs are for the category of language features. Most of the convergent KUs are categorized as KUs for data types (30%) and language features (60%),

[14]For reference, we also show KUs that were not categorized as absent, divergent, or convergent according to the cuttoff points shown in Table 5.

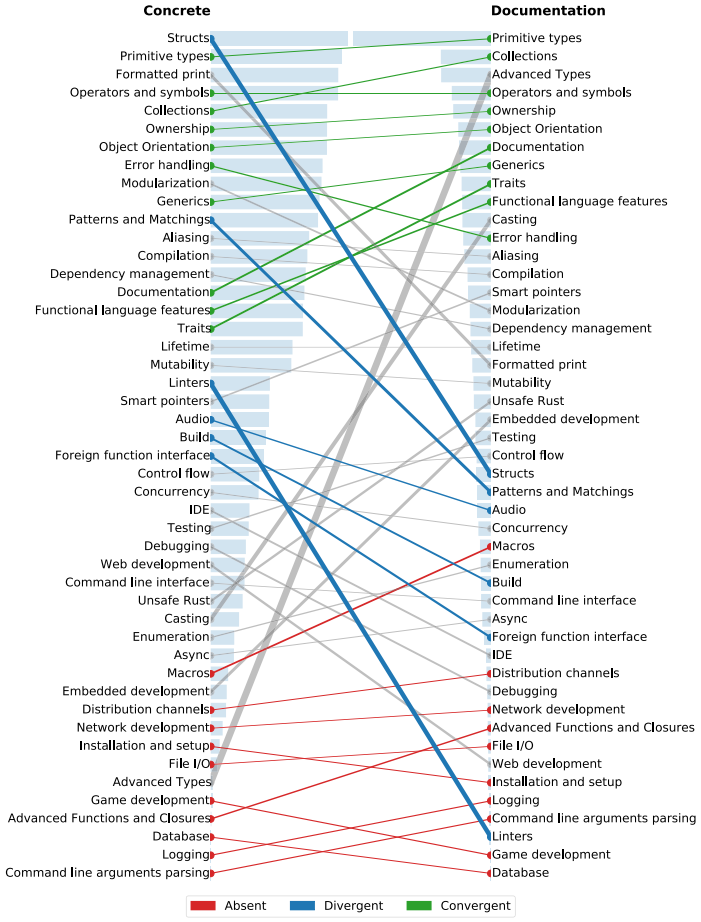


Fig. 8. The absent, divergent, and convergent KUs in terms of the frequency metric. The bars indicate the frequency of each KU.

suggesting that documenters do emphasize elementary KUs that occur frequently in the concrete documentation model. Table 6(b) shows the proportion of absent, divergent, or convergent KUs grouped by the category of the KUs. We can observe that the category of KUs for programming niche has a high proportion of absent KUs (41.6%). In addition, the category of KUs for data types has a high proportion of convergent KUs (42.8%). Also noteworthy is the proportion of divergent KUs categorized as language features (33%), suggesting that the official Rust documentation should consider improving the coverage of this category of KUs.

Confirmatory analysis: Table 7 shows the results of our validation of the absent and divergent KUs that were identified by our approach. For each of the absent and divergent KUs, the table records the information sources where supporting evidence of topical misalignment was identified. We can observe that, among the absent KUs, four KUs (for distribution channels, game development, logging, and command line arguments parsing) do not have any piece of supporting evidence of topical misalignment. Similarly, among the divergent KUs, the only KU for which we did not identify any piece of evidence is the KU for audio. Interestingly enough, should we have chosen slightly more conservative cutoff points (see Table 5), our approach would have not deemed the KU for audio as divergent. We can also observe a difference in the number of extracted quotes

Table 6. The Relationship between the Proportion of Each Type of KU Alignment and the Proportion of Different Categories of KUs

(a) Proportion of KUs in each category of KUs grouped by the type of KU alignment.

KU alignment	Category of KUs	Proportion of KUs
Absent	Development tooling	10.0%
	Language features	40.0%
	Programming niche	50.0%
Divergent	Data types	16.7%
	Development tooling	33.3%
	Language features	33.3%
	Programming niche	16.7%
Convergent	Data types	30.0%
	Development tooling	10.0%
	Language features	60.0%

(b) The proportion of absent, divergent, and convergent KUs grouped by the category of KUs.

Category of KUs	KU alignment	Proportion of KUs
Data types	Divergent	14.2%
	Convergent	42.8%
Development tooling	Absent	10.0%
	Divergent	20.0%
	Convergent	10.0%
Language features	Absent	22.0%
	Divergent	33.0%
	Convergent	11.0%
Programming niche	Absent	41.6%
	Divergent	8.3%

Table 7. Results of the Validation of the Absent and Divergent KUs

KU	Information source			Quotes
	IRs/PRs	Surveys	Q&A posts	
Structs	✓	✗	✓	Q1, Q2, Q3, Q4, Q5, Q6, Q7
Patterns and matching	✓	✗	✗	Q8, Q9, Q10, Q11, Q12
Linters	✓	✗	✗	Q13, Q14, Q15, Q16, Q17, Q18, Q19
Audio	✗	✗	✗	–
Build	✓	✗	✓	Q21, Q22, Q23, Q24, Q25, Q26
Foreign function interface	✓	✗	✓	Q29, Q30, Q31, Q32, Q33, Q34
Macros	✓	✗	✓	Q35, Q36, Q37, Q38, Q40, Q40, Q41, Q42, Q43, Q44, Q45, Q46, Q47, Q48
Distribution channels	✗	✗	✗	–
Network development	✗	✗	✓	Q49
Installation and setup	✓	✗	✗	Q50, Q51
File I/O	✗	✗	✓	Q52, Q53
Game development	✗	✓	✗	–
Advanced functions and closures	✓	✗	✓	Q54, Q55, Q56, Q57, Q58, Q59, Q60, Q61
Database	✗	✗	✓	Q62, Q63
Logging	✗	✗	✗	–
Command line arguments parsing	✗	✗	✗	–

A check mark (✓) indicates that a piece of evidence was gathered from the respective information source for a certain KU. Conversely, a cross mark (✗) indicates that no evidence was gathered from the respective information source and KU.

among the KUs, which suggests that some KUs attract more voluntary statements from developers about topical misalignment than others. The verbatim quotes can be found in Appendix D. Based on our confirmatory analysis, we can validate the existence of at least six absent KUs for macros, network development, installation and setup, file I/O, advanced functions and closures, and database. In addition, we can validate the existence of at least five divergent KUs for structs, patterns and matchings, linters, build, and foreign function interface.

How many KUs are absent, divergent, or convergent in the Rust documentation?

Our approach identifies 6 divergent KUs, for which 5 can be verified through a qualitative analysis of external information. Our approach also identifies 10 absent KUs in terms of the frequency metric, half of them categorized as KUs for programming niches (e.g., network, game, and database development). A total of 6 KUs that are identified as absent can be verified through qualitative analysis of external information. Also, one-third of the KUs that are categorized as language features by our approach are divergent.

5.3 Do Convergent, Divergent, and Absent KUs Exhibit Different Awareness in Q&A Websites?

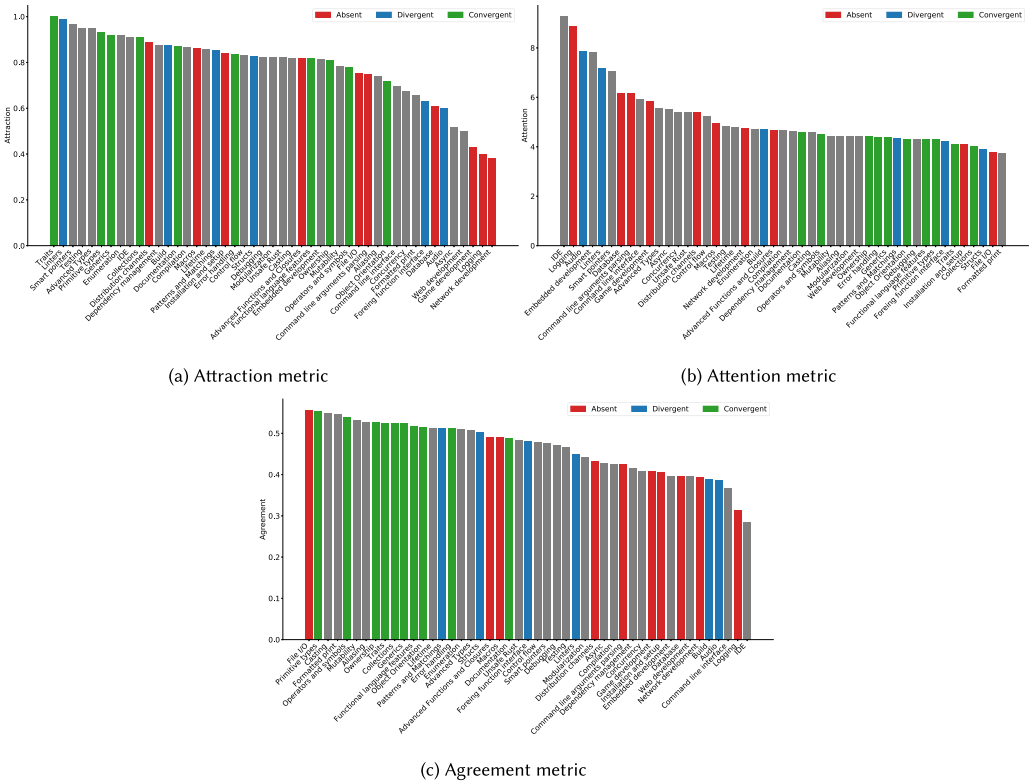
Motivation: Documenters will benefit from recognizing KUs that need to be prioritized by their documentation efforts. By triangulating awareness metrics from the concrete documentation model with the types of KU alignment, documenters can reason about KUs that should be prioritized in the documentation. In particular, documenters can assess the extent to which developers can find information about convergent, divergent, and absent KUs in the form of answers to Q&A posts and, based on this assessment, decide how to prioritize certain KUs to update the maintained documentation.

Approach: We initially calculate the three awareness metrics for all KUs, from which we obtain three ranked lists of KUs and their respective median attraction, attention, and agreement. We then use the Spearman ρ rank correlation to verify whether there is any significant difference between each pair of awareness metrics that are calculated for the KUs (the estimated *rho* coefficient is interpreted as described in Section 5.1.1).

Next, we categorize each KU according to their type of KU alignment (see Section 5.2) and calculate the median of each metric for each of the three types of KU alignment. We then compare the rank of the absent, divergent, and convergent KUs for the same metric. Our comparison consists of using the Mann-Whitney procedure ($\alpha = 0.05$) to perform a post hoc hypothesis testing of the difference between one type of KU alignment (e.g., convergent KUs) and another type of KU alignment (e.g., divergent KUs) for a given awareness metric. For each of the awareness metrics, we test three hypotheses that compare convergent vs. divergent KUs, convergent vs. absent KUs, and divergent vs. absent KUs. After observing the median of each awareness metric for each type of KU alignment, we decide whether to choose a one- (difference between medians is less than 5%) or two-sided (difference between medians is greater or equals to 5%) test and, when a two-sided is chosen, the direction of the test. We correct the multiple hypothesis tests using the Bonferroni procedure [50] over the groups of equivalent comparisons across the different metrics. For instance, we correct the three tests for the convergent vs. divergent comparison across the attraction, attention, and agreement metrics. For each rejected null hypothesis, we calculate the magnitude of the difference using the Cliff's Delta estimator of effect size, with the same interpretation of the estimates as described in Section 5.2.

Results: Three out of four KUs with the lowest rank for attraction are absent in the Rust documentation, indicating KUs that occur in few documents and often remain unanswered. Figure 9 shows the ranked KUs by their attraction, attention, and agreement metrics. We observe a relatively uniform distribution of values for the same metric across the different KUs. The median attraction, attention, and agreement are, respectively, 82.2%, 4.66, and 48.0%. The median value of each metric grouped by type of KU alignment is shown in Table 8(a). In terms of correlation between the awareness metrics, the only correlated rank of KUs is between the attention and agreement metrics ($\rho = -0.653$, moderate) with the correlation estimate suggesting a decreasing monotonic relationship between the ranks (i.e., while the rank of a KU is high for the attention metric, the rank for the same KU is low for the agreement metric and vice versa). The correlation between the rank of KUs for the attraction and attention, as well as for the attraction and agreement, is not statistically significant ($\alpha = 0.05$), suggesting that these metrics do not share common factors associated to their values.

Observing Figure 9(a), **there is no evident pattern that separates absent, divergent, and convergent KUs in the rank of KUs by the attraction metric.** Nonetheless, the rank of KUs by the attention and agreement metrics presents clear yet contrary patterns. Figure 9(b) suggests that **the attention metric tends to be larger for absent and divergent KUs than for**



(a) Attraction metric

(b) Attention metric

(c) Agreement metric

Fig. 9. KUs ranked by their attraction, attention, and agreement metrics.

convergent KUs, whereas the difference between absent and divergent KUs is not apparent. In turn, Figure 9(c) suggests that **the agreement metric tends to be larger for convergent KUs than for absent and divergent KUs**, while the difference between absent and divergent KUs is also unclear. Table 8(b) shows the results of our hypothesis tests, which partially confirm our initial observation of Figure 9. Particularly for the attraction metric, the difference between the rank of absent, divergent, and convergent KUs is not statistically significant. Also, divergent and absent KUs have no statistically significant differences for any of the awareness metrics. However, the difference between convergent and absent KUs is statistically significant for both the attention and agreement metrics, as well as the difference between convergent and divergent KUs with respect to the agreement metric. All the statistically significant differences have a large effect size.

The observation that the attention of convergent KUs is systematically smaller than that of absent KUs indicates that convergent KUs typically occur in Q&A posts that receive fewer answers. However, by definition, convergent KUs are also associated with a larger proportion of the documentation. The high amount of associated documents can be related to the fact that most of such KUs are for data types and language features, i.e., they are elementary KUs (see Section 5.1.1) that are required by almost every Rust developer and related to many other KUs. In turn, the low amount of answers in Q&A posts can be related to the fact that information about such elementary KUs occurs with a high frequency in the concrete documentation and, therefore, generates fewer discussions in the form of answers in Q&A posts. The aforementioned conjectures about the attention of convergent and absent KUs are supported by triangulation with the agreement of those two KU alignments. The agreement for convergent KUs is significantly larger than for absent (and

Table 8. The Relationship between Awareness Metrics and Types of KU Alignment

(a) Median attraction, attention, and agreement metrics for absent, divergent, and convergent KUs.

KU alignment	Awareness metric	Median
Absent	Attraction	75.0%
	Attention	5.17
	Agreement	41.0%
Divergent	Attraction	84.0%
	Attention	4.53
	Agreement	46.0%
Convergent	Attraction	85.0%
	Attention	4.36
	Agreement	52.0%

(b) Rank-based comparison between absent, divergent, and convergent KUs for attraction, attention, and agreement metrics.

Awareness metric	Null hypothesis (H_0)	Reject H_0?	Cliff's Delta (d)
Attraction	Convergent = divergent	No	—
	Convergent ≤ absent	**Yes**	0.56 (large)
	Divergent ≤ absent	No	—
Attention	Convergent ≥ divergent	No	—
	Convergent ≥ absent	**Yes**	−0.62 (large)
	Divergent = absent	No	—
Agreement	Convergent ≤ divergent	**Yes**	0.90 (large)
	Convergent ≤ absent	**Yes**	0.76 (large)
	Divergent ≤ absent	No	—

divergent) KUs, suggesting that developers are more likely to accept an answer for a post that is related to a thoroughly documented KU.

Do convergent, divergent, and absent KUs exhibit different awareness in Q&A websites?

Absent and convergent KUs tend to have opposite patterns of attention and agreement. Absent KUs have high ranked values for the attention metric and low ranked values for the agreement metric. Conversely, convergent KUs have low ranked values for attention and high ranked values for agreement. A statistically significant difference between the ranks of KUs for the divergent and convergent KUs concerning the agreement metric exists. All these differences regarding the ranks of KUs have a large effect size.

6 DISCUSSION

This section discusses the results described in Section 5. We draw practical implications for documenters of programming languages and externally validate our results. In Section 6.1, we discuss the usage of our approach by documenters that want to decide what topics of the documentation should be prioritized when updating the documentation. In Section 6.2, we report on our experience and estimate the effort to instantiate our approach. In Section 6.3, we discuss how documenters can leverage our approach to make informed decisions about the design of data collection instruments to assess the alignment of a programming language documentation (e.g., survey). In Section 6.4, we discuss the different maintenance types that we observed being applied to the Rust documentation. In Section 6.5, we discuss our observations with respect to the root causes of the misalignment between documentation and information needs of developers. Finally, in Section 6.6, we compare our findings with those of prior surveys run by the Rust community.

6.1 Prioritization of Documentation Contents and Topical Alignment

The generated results by applying our approach can be leveraged by documenters to identify KUs that are not well covered by the documentation and decide which of such KUs to prioritize when updating the documentation. For instance, an analysis of the absent KUs suggests documentation topics that should be improved, as absent KUs lack in both the concrete and the documentation models. To decide which absent KUs should be prioritized when incorporating new information into the documentation, documenters can search for absent KUs that often attract responses in Q&A posts (i.e., high attraction or attention) but still have a low rate of agreement. One example

is the KU for distribution channels that, although being highly ranked in terms of attraction, is low ranked in terms of agreement. The same triangulation can be performed for divergent KUs. When deciding which divergent KUs to prioritize, documenters should consider the rank of the awareness metrics associated with those KUs. For example, documenters should consider prioritizing divergent KUs with higher levels of attraction or attention and lower levels of agreement, as information seekers will find it challenging to obtain accepted answers about divergent KUs from Q&A posts. One example is the KU for build, which is associated with high attention but still exhibits low agreement. In addition, by analyzing the agreement metric of convergent KUs, documenters can spot information that, although widely covered by the official documentation, has a high number of posts that remain unanswered in Q&A websites.

6.2 The Required Effort for Instantiating Our Approach

In Section 4.1, we describe in detail the performed steps to instantiate our approach to assess the topical alignment of Rust documentation. Here, we reflect on the effort to perform such an instantiation, with the objective of providing an estimate of the effort that should be spent for other programming languages. Although our approach can be almost entirely automated, some steps require specific manual interventions or the acquisition and pre-processing of external data. The two related steps with the modelling of domain knowledge from the language documentation require most of the involved manual effort, namely, the extraction of anchor words and KUs (Section 4.1.1) and the domain expert review (Section 4.1.2). However, depending on the expertise of the documenter that performs the former step, the latter step might be dispensed (e.g., the person performing the extraction of anchor words and KUs is the same person that writes the documentation). As a reference, the extraction of anchor words and KUs from the official Rust documentation by someone without any background on the language required approximately 20 person-hours, while the domain expert review required approximately 1 person-hour. Most of the time during domain knowledge modelling was spent on understanding the language concepts, coming up with representative anchor words for the learned concepts, and grouping the related anchor words into KUs. Therefore, the required effort for domain knowledge modelling can be drastically reduced should the person performing the KUs and anchor words extraction be experienced in the programming language and its concepts. The effort for domain knowledge modelling also depends on the granularity level with which the anchor words and KUs describe the contained knowledge on the documentation. The more broad, detailed, and fine-grained the domain knowledge model is, the more effort is likely to be required. We instantiated our approach considering a wide coverage of the documentation contents while avoiding overspecific anchor words, as explained in Section 4.1.1. Finally, the organization level and the documentation style also correlate with the required effort for domain knowledge modelling. The more compartmentalized the explained concepts in the documentation are, the easier it is to derive representative anchor words and to group them into KUs. For instance, the Rust documentation is organized into thematic books, which are in turn divided into chapters that present the concepts in a structured way. This form of organization of the documentation facilitates domain knowledge modelling, compared with less structured documentation.

The related steps with topic modelling and topical alignment measurement can be entirely automated, enabling users to iteratively refine the assessment of the topical alignment of a programming language's documentation. Nonetheless, both the collection and pre-processing of external data, as well as the set up of parameters, are required to automate our approach. As for the collection and pre-processing of external data (Sections 4.1.3 and 4.1.4), there are important requirements to enable the deployment of our approach. Since the contents of Q&A websites are updated at a fast rate, it is important to set up a streaming infrastructure to periodically acquire new data that

reflect the most up-to-date needs of developers. As the data from the stream gets in, the posts of interest need to be selected so the contents of the selected posts are pre-processed. We learned from our experience in instantiating our approach to Rust that the automated selection of posts of interest (described in Section 4.1.3—Select Rust-related Posts of Stack Overflow) can be significantly accelerated should these posts be indexed by their tags. As for the setup of parameters, when automatically selecting posts of interest (i.e., Rust-related posts of Stack Overflow), we have successfully adopted the same TET and TST parameters as reported by prior work (see Section 4.1.3—Select Rust-related Posts from Stack Overflow). Hence, it is likely that the same parameters can be successfully reused for the instantiation of our approach for other programming languages as well. During topic modelling (described in Section 4.1.3—Topic Modelling), the parameters for the number of topics and the anchor strength need to be fine-tuned, which might be done through manual analysis of the built topic models. Nonetheless, the calculation of the model's overall TC can be used to automatically fine-tune the determination of the number of topics (by searching for the number of topics that optimizes TC). Similarly, classical coherence metrics for the evaluation of topic models can be leveraged to automate the determination of the anchor strength (by searching for the anchor strength value that optimizes the topics' coherence).

6.3 Informing Documentation Surveys

When designing instruments for collecting data from the documentation users (e.g., surveys), documenters can leverage our approach to decide with more confidence which questions to emphasize. One of the approaches used by documenters to understand the opportunities to improve a programming language's official documentation is to release surveys to developers. Despite the accurate, detailed, and significant information received through surveys, collecting and analyzing data using this instrument can require high amounts of time and effort. For example, the generated data during the design of the latest Rust Annual Survey [42] shows that running the survey involved four-month-long teamwork [43], including identifying key stakeholders [35], designing the survey questionnaire [31, 32], translating the questionnaire to multiple languages [43], obtaining feedback from the authors of prior surveys [36], hosting the survey in a specialized platform [34], and analyzing and publicizing the results [33]. The effort to run the survey is such that the responsible team for its design explicitly questions its cost-effectiveness: "(...) we should ask if the survey in its entirety is producing enough useful data to be worthwhile (given the relatively high cost of producing the survey, analyzing the data, etc.)" [31]. One important lesson learned by applying our approach to assess the topical alignment of Rust documentation is that it is relatively easy to obtain data to build a concrete model of the information needs of developers. After setting up a basic infrastructure for data collection and model training, documenters can efficiently perform iterative analyses to validate and refine hypotheses and complement the results of their survey efforts. For example, consider a closed question of the latest survey released by the Go community that asks "when using official Go documentation, have you struggled with any of the following." Among all answers for this question (n = 2,476), 31% selected "none of the statements apply to me," showing that documenters will benefit from having a form of iteratively refining research questions. Furthermore, documenters can vary the form of domain knowledge specification (e.g., by considering different KUs or associating different anchor words to the existing KUs), such that different KUs of interest can be incorporated into the documentation models, and new analysis can be performed about more specific or general topics. As another example of the Go survey, the question "how helpful is official Go documentation for achieving your programming goals in the following areas" contains a set of seven pre-specified answers. Based on the obtained responses, documenters can use our approach to further understand the documentation of related KUs with the "least helpful" areas. More concretely, in the example question of the Go survey, the most

popular answer was "using modules." Documenters can then hypothesize about a set of modules that developers might be encountering difficulty finding documentation about and add those modules as KUs to a new iterative run of our approach. Nonetheless, our approach is not intended to replace surveys as a form of assessing developers' information needs but, rather, to complement this instrument. In this sense, much of the analyses of the KU metrics can be used to either inform the design of the documentation or to inform the development of other data collection instruments, such as surveys.

6.4 On the Maintenance of the Documentation

In Section 5.2, when analyzing the selected quotes for each of the absent and divergent KUs (see Appendix D), we can observe that different maintenance types are applied to the documentation. More specifically, adopting the software maintenance dimensions as proposed by Swansom [90], we observe that corrective, adaptive, and perfective maintenance are equally applied to the documentation of those KUs. This observation suggests that, in parallel to the addition of new contents, absent and divergent KUs should also require all types of maintenance efforts from documenters. Therefore, documenters should carefully balance the allocation of resources to the addition of new content and the maintenance of the existing contents for absent and divergent KUs. Furthermore, while corrective and perfective maintenance is typically performed on the basis of distributed and isolated contributions from the developers that use the documentation (e.g., by means of IRs and PRs), adaptive maintenance and new contents addition requires a coordinated effort from the core documentation contributors to keep track of new languages features and document them accordingly.

6.5 Leveraging Co-frequency to Meet Developers' Information Needs

During our qualitative analysis described in Section 5.2, we observe that topical misalignment can stem either from both the documentation contents or the documentation presentation as well. For example, occasionally developers report that their information needs cannot be met due to issues with the documentation searchability [2, 3] and navigability **??**. In Section 5.1.2, we also found that multiple KUs typically co-occur in the same document, which suggests that documenting information about Rust often requires the compilation of multiple subjects in a single document. We also compared the distribution of the number of KUs that co-occur in the same document between the concrete and documentation models. We found that a large difference exists between the distributions. These observations suggest that documenters should carefully consider how KUs should be combined in the same document to match developers' information needs. A possible solution that documenters can consider to improve documentation searchability and navigability is cross-linking related KUs in the official documentation. The distribution of the co-frequency and affinity metrics of the concrete model can be used to assess which required information by developers typically involves more than one KU. The co-frequency and affinity metrics can highlight pairs of KUs for which documenters should pay attention to the differences and similarities and potentially cross-link these KUs. For instance, by manually inspecting the ranked list of pairs of KUs with the highest affinity, we observe that the KUs for traits and object orientation have a high affinity. Although this observation is not surprising, given that both KUs share a remarkably high number of related concepts, in fact, traits do not support all features of an object in a strict sense. In particular, the dynamic binding mechanism of traits does not support the concept of inheritance typically found in languages designed as object-oriented (e.g., Java). Therefore, pairs of KUs with high co-frequency and affinity can help documenters to identify KUs for which developers will benefit from examples that promote the comparison between related concepts of one KU

to another, as well as KUs whose differences and similarities should be better emphasized in the documentation.

6.6 External Validation

In Section 5.1, we found that the rank of prevalent KUs is similar between the concrete and documentation models. This observation suggests that, with some exceptions, developers seeking for information will find similar distributions of KUs among posts of Q&A websites and the official documentation. Still, it is valuable to pay attention to the absent and divergent KUs, as they represent topical misalignments. Table 6(a) shows that the associated KUs with programming niches are at a higher risk of being absent from the Rust documentation. In addition, a close inspection in the ranked KUs by attraction shows that the KU for game development receives fewer answers than other absent KUs. This observation can be externally validated by examining the Rust Game Development Survey released in 2020 [8]. When asked about the negative aspects of using Rust for game development, respondents often point to documentation as one of these aspects, which confirms that developers do have an interest in improving the documentation related to the KU for game development. It is also noteworthy the willingness of the Rust community to promote the adoption of the language for other niches that are associated with absent KUs. For example, a working group for database development (one of the absent KUs) was recently created, and the Rust users already publicly expressed their wish for documentation improvement: "*I'd love to see some 'database book' that covers connections, drivers, pools, has examples for every feature and gives overview of ecosystem. Individual project would benefit greatly from better docs*" [5]. These observations suggest that engaged communities promoting the adoption of Rust for programming niches that are associated with absent KUs should attract documenters. Interestingly, some well-documented programming niches (e.g., embedded development, which has a specialized book in the official documentation [17]) are not deemed as divergent nor absent, suggesting that these niches are likely well prepared for a potentially growing community of developers.

In the past few years, the Rust community ran the Rust Survey [7] to understand the evolution, challenges, and opportunities of the language. One of the survey questions regards the difficulty of specific topics. Participants have pointed out that lifetime, ownership, macros, trait bounds, and async are the most "tricky or very difficult" topics, respectively. We observe that, while the KUs for traits and ownership are perceived as difficult, they are also associated with convergent KUs. A plausible relationship between the perceived difficulty of such KUs and their high frequency on Q&A websites and the documentation is that information about those KUs might be noisy and often scattered over many documents, making it difficult for developers to pinpoint the information that discusses specificities of these KUs. Therefore, documenters could consider new forms of organizing documentation regarding those KUs, focusing on understanding specific pain points from developers on those KUs, and tackling more specific issues around those KUs in the documentation. Additionally, the KU for macros is both absent and perceived as difficult by developers, raising a flag to documenters regarding the quality and completeness of the documentation around this specific KU.

7 THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

Selection of Rust-related Posts from Stack Overflow: As we leverage the tags from Stack Overflow to identify Rust-related posts, our set of posts can be incomplete (i.e., we might have missed some Rust-related posts). The incompleteness stems from the fact that some users might use an unrelated tag to Rust that our selection procedure did not capture. Nonetheless, we expect

that this threat is satisfactorily mitigated by expanding the list of intially selected posts using the TET and TST thresholds.

Anchor Words Definition: Our set of anchor words was directly derived from the official Rust documentation. Still, the set of anchor words was used to train topic models from data from Q&A websites. Therefore, some of the adopted terminology can differ between the two data sources. Nonetheless, to mitigate this threat, we use the all known related terms to compose our set of associated anchor words with a KU and validated the terminology with a domain expert.

KU Metrics: A fundamental principle to the validity of our metrics (i.e., the ability of our metrics to capture the prevalence and awareness of a KU) is the quality of the topic-document assignment of our models. With regards to this factor, during our analyses, we performed sanity checkings to ensure that the topics assigned to each post were in fact corrected. We observe that our topic models can reliably determine the set of topics in a post. However, we also observe that the ability of the dominant topic to distinguish the main subject of a post is smaller. We also verified the ability of our metrics to measure distinct phenomena by running different correlation analyses. Although some degree of correlation exists between our metrics, we can verify that these metrics are able to measure different aspects related to the prevalence and difficulty of a KU.

Posts without Topics: Anchored CorEx assigns an individual probability for each topic in a post. Since all assigned topic probabilities to a post can be lower than 0.5, some posts can end up without any occurring topics after calculating the occurrence metric (see Section 4.2.1). To understand how this scenario might threaten the validity of our results, we investigated how often it happens and we found that only 0.43% of the posts have no occurring topics, which has a negligible impact on our results.

Definition of Absent, Divergent, and Convergent KUs: To categorize KU into the three types of KU alignment, we adopted a cutoff threshold to separate the rank of KUs into segments. We experimented with many different combinations of cutoff values in search for a configuration that allowed us to obtain interpretable and presentable results. For example, during our experimentation, we searched for a cutoff setup that included at least one of the three types of KU alignment and that included a certain number of KU in each type such that we could discuss the results in the article. Nonetheless, documenters interested in using our approach can experiment with other cutoff values, depending on how conservatively they want to represent the three types of KU alignment.

8 CONCLUSIONS

In this article, we present a machine learning–based approach to study the topical alignment of programming languages documentation, defined as the difference between the concrete developers' information needs and the current state of the documentation. Our approach uses a non-parametric semi-supervised topic model to derive the set of topics from posts of Q&A websites. Such topic model represents the topics that make up the concrete information needs of developers, and it is used to derive the topics of the programming language documentation. We also propose a set of metrics to measure and evaluate the differences and similarities between the topics of the Q&A websites and the topics of the programming language documentation. Based on our approach, we perform an empirical study using Rust as a case study.

Our empirical study reveals that, in general, the distribution of topics that make up the concrete information needs of developers share a high degree of similarity with the topics of Rust documentation. Nonetheless, our study highlights topics for which the Rust documentation requires improvement and indicates missing topics from the documentation that developers are not able to obtain answers from Q&A websites. More generally, we identified that KUs that occur with less frequency in both the Q&A websites and the official documentation are often associated with KUs

that are categorized as programming niches (e.g., network, game, web, and database development). Those KUs also have a high number of answers in Q&A websites but, at the same time, have lower odds of having an accepted answer. In addition, KUs that are categorized as language features (e.g., structs, patterns and matchings, and foreign function interface) often occur in Q&A websites but not in the official documentation of Rust. These observations are noteworthy, as they suggest KUs that should receive priority from documenters when updating the documentation. Our study contributes to the body of knowledge regarding the documentation of programming languages and is of particular interest for practitioners that are engaged in actively producing information instead of passively consuming information from the documents collection.

APPENDICES

A DESCRIPTION OF THE KUS

Table A.1 shows the extracted KUs from the official documentation of Rust.

Table A.1. The Derived KUs from Rust's Official Documentation and Their Respective Anchor Words

Category	Knowledge Unit	Anchor words
Data types	Primitive types	type, primitive_type, scalar_type, compound_type, tuple, array, scalar, slice
	Structs	struct, named_field, name_field, structure
	Enumeration	enum, enumeration, variant
	Collections	common_collections, common_collection, collections, collection, vector, hashmap, hash_map, append, index, std_collection, stdcollection, std_collections, stdcollections, vec, hashset, hash_set
	Generics	generic, generic_type, concrete, abstract, monomorphization, bound, phantom_type
	Smart pointers	smart_pointer, reference_counting, deref, deref_trait, drop_trait, interior_mutability, reference_cycles, cons_list, con_list, cons_function, con_function, reference_cycle, recursive_type, dereference_operator
	Advanced Types	newtype, newtype_pattern, type_alias, never_type, dynamically_sized_type, dst, unsized_types, dynamic_size_type, dynamic_size, unsize_type, unit_type
Development tooling	IDE	rls, rust_analyzer, ide, visual_studio
	Linters	rustfmt, rustfix, clippy, linter, lint, rust-clippy
	Installation and setup	installation, install, hello_world, rustup, rustup.rs
	Modularization	module, reuse, external_code, private, public, visibility, file_hierarchy, file_hierarch, reexport, workspace
	Testing	test, automated_test, automate_test, assert, unit_test, integration_test, cargo_test
	Dependency management	release, release_profile, library, crates_io, yank, cargo_install, lockfile, dependency, update_dependency, cargo_publish, semver
	Compilation	rustc, crate, compilation_unit, crate_file, compiler, compilation, flag, compiler_flag, compile_flag, compiler_option, compile_option, cargo_rustc, crate_type, link_crate, abi, application_binary_interface
	Build	build, cargo_toml, build_script, build_rs, cargo_build
	Documentation	rustdoc, cargo_doc, cargodoc, doctest, document_api, rfc1574, rfc1946, pass, passes
	Debugging	debug, std_fmt, debug_info, debug_symbol, debugger, rustdt, msvc_abi, rust_gdb, gdb, rust_lldb

(Continued)

Table A.1. Continued

Category	Knowledge Unit	Anchor words
Language features	Ownership	ownership, stack, heap, reference, borrow, move, copy, partial_move, drop, forget
	Error handling	error_handling, exception, recoverable_error, unrecoverable_error, unrecover_error, panic, abort, unwind, result, option
	Functional language features	functional_programming, functional_style, closure, anonymous, anonymous_type, anonymous_function, type_anonymity, type_anonym, iterator_trait, next_method, hof, high_order_function
	Object Orientation	object_oriented, object_orientation, oo, oop, object, inheritance, trait
	Patterns and Matchings	pattern_matching, pattern_match, match, if_let, while_let, matches
	Unsafe Rust	unsafe, unsafety, memory_unsafety, unsafe_trait, raw_pointer, transmute
	Traits	advanced_trait, associated_types, operator_overload, default_type_parameter, trait_implementation, implement_trait, supertrait
	Advanced Functions and Closures	function_pointer, function_point, return_closure, returning_closure, hrtb, higher_rank_trait_bound
	Macros	macro, declarative_macro, procedural_macro, metaprogramming, derive_macro
	Operators and symbols	operators, symbols, literal, constant, const
	Distribution channels	nightly, night, distribution_channel, distribut_channel, night_rust, train_schedule, rust_release, next_train, release_train, feature_flag, install_night, unstable_feature, nightly_release, night_release
	Mutability	variable_binding, mutable, mutability, immutable, mut, mut_modifier, freeze, frozen
	Lifetime	variable_shadow, shadow, variable_binding, block, scope, raii, lifetime, drop, scope, out_scope, lifetime_annotation
	Casting	cast, casting, type_conversion, type_inference, from_trait, into_trait, tryfrom, tryinto, tostring, fromstr
	Aliasing	alias, type_alias
	Control flow	condition, boolean_condition, boolean_condit, control_flow, decision, loop, break, continue, std_iter, named_loop
	File I/O	file_read, file_write, file_method, read_only, readonly, file_descriptor, write_only, writeonly, line_file, std_io, open_options
	Foreign function interface	ffi, foreign_function_interface, foreign_interface, foreign_function, foreign_library, snappy, libc, wrap, foreign_global, no_mangle, bindgen
Programming niche	Concurrency	concurrency, concurrent, race_condition, deadlock, multithread, thread, process, spawn, message_pass, producer, consumer, mutex, mutexes, std_sync, std_thread, rayon
	Game development	game, roguelike, sokoban, game_engine, windowing, physics, rendering, fmod, openal, ggez

(Continued)

Table A.1. Continued

Category	Knowledge Unit	Anchor words
	Web development	web, web_development, web_api, rocket, actix_web, diesel, sqlx, oauth, cookie, oauth2, html, css, webassembly, http, http_request
	Network development	openssl, network, network_layer, dns, tcp_ip, ip, tcp, udp, std_net, ftp, smtp, cryptography, encryption, certificate
	Embedded development	embed, embedded, microcontroller, qemu, interrupt, spi, uart, rs232, usb, i2c, ttl
	Database	database, sqlx, db, mongodb, sqlite, postgres, mysql, sql, driver, orms, orm, elasticsearch
	Logging	logging, log, console_logger, env_logger, log4rs, fern, syslog, log
	Audio	audio, midi, music, audio_decoder, audio_decode, audio_encoder, audio_encode, libsoundio
	Command line interface	command_line, argument_parse, ansi_terminal, ansi_term, terminal, ansi, termcolor
	Formatted print	format_print, formatted_print, formating, format, print, formatting_trait, format_trait
	Async	tokio, async_std, mio, future
	Command line arguments parsing	clap, structopt, std_env_arg

B INITIAL TAG SET

Table B.1 shows the initial set of Rust-related tags.

Table B.1. Our Initial Set of Rust-Related Tags

Category	Tag	
	Derived from Stack Overflow	**Derived from documentation**
General	"rust"	–
Language feature	"rust-macros"	"borrow"
	"rust-proc-macros"	"borrow-checker"
	"rust-obsolete"	"supertrait"
	"rust-crates"	"unsafe"
	–	"refcell"
	–	"collections"
	–	"crate"
Development tool	"rust-cargo"	"rustc"
	"rust-tokio"	"rustup"
	"rust-diesel'	"rustdoc"
	"rust-actix"	"rustfix"
	"rust-rocket'	"clippy"
	"rust-warp"	"rls"
	"rust-chrono"	–
	"rust-piston"	–

C DERIVED TOPICS

Table C.1 shows the top 20 words of the semi-supervised topic model associated with the concrete model.

Table C.1. The Top 20 Words of Each Derived Topic of the Concrete Model

Topic (associated KU)	Word types
Advanced Types	unstable, newtype, specialization, dst, type_alias, dynamically_sized_type, unsized_type, unit_type, unboxed_closures, fn_traits, const_fn, unsize, type_alias_impl_trait, never_type, newtype_pattern, type_ascription, associated_type_defaults, dynamic_size, specialized, newtypes
Collections	vec, vector, iter, map, hashmap, element, item, collection, std_collection, index, string, hashset, collect, hash_map, push, iterator, len, into_iter, append, usize
Enumeration	enum, variant, enums, enumeration, myenum, union, discriminant, tag, untagged, e0015, nil, exhaustive, tagged, strum, variant1, strum_macros, variant2, uninhabited, discriminator, boolean
Generics	trait, type, generic, parameter, bound, concrete, generic_type, abstract, note, sized, monomorphization, required, requirement, ops, constraint, known, annotation, inference, mismatch, accept
Primitive types	value, type, element, vector, slice, array, tuple, scalar, primitive_type, length, integer, indexing, tuples, val, f64, pair, f32, arr, contiguous, elem
Smart pointers	deref, interior_mutability, derefmut, case, smart_pointer, actually, mean, memory, doe, reason, example, different, instead, data, reference_counting, possible, kind, understand, fact, place
Structs	struct, structs, structure, impl, self, fn, pub, field, deserialize, foo, phantomdata, member, serialize, mystruct, bool, eq, name_field, constructor, named_field, u32
Build	build, cargo, cargo_toml, cargo_build, build_script, project, toml, github, registry, building, git, 1ecc6299db9ec823, built, manifest, edition, latest, profile, builder, ci, cdylib
Compilation	build, cargo, rustc, compilation, compiled, flag, compile, crate, library, abi, crate_type, cargo_rustc, compiler, compiling, v0, aborting, compiler_flag, previous, information, verbose
Debugging	debug, std_fmt, gdb, debugger, derive, debug_info, partialeq, fmt, breakpoints, debuginfo, formatter, rustdt, unoptimized, display, rust_gdb, debug_symbol, serde, codelldb, debugging, openocd
Dependency management	library, cargo, dependency, release, package, crate_io, semver, standard, version, cargo_install, author, lockfile, yank, party, publish, compatibility, update_dependency, published, utils, publishing
Documentation	test, pas, passed, passing, pass, doctest, cargo_doc, run_test, utilizing, new_string, testresult, arg3, arg2, mycallback, redacted, den, d4h0, spice, fluff, alleviates
IDE	ide, vscode, rls, intellij, visual_studio, thanks, lot, right, maybe, help, post, yes, thank, language, issue, people, try, great, guess, thought
Installation and setup	install, installed, hello_world, rustup, installation, x86_64, gnu, usr, toolchain, toolchains, rustlib, unknown, ld, libstd, pc, installing, libcore, cc, msvc, ubuntu
Linters	closed, topic, reply, day, automatically, wa, allowed, longer, clippy, lint, new, invite, rustfmt, comment, open, question, hello, linting, linter, alice
Modularization	crate, module, import, public, private, workspace, exported, visibility, submodule, submodules, publicly, privacy, mod, reexport, external, export, unresolved, importing, imported, external_code
Testing	test, assert, unit_test, cargo_test, it_works, integration_test, assert_eq, my_test, cfg, testing, should_panic, test_foo, super, bench, some_test, assertion, bencher, integration, measured, suite
Advanced Functions and Closures	function_pointer, hrtb, return_closure, returning_closure, higher_rank_trait_bound, function_point, call_it, add_closure, higher_order, amet, dolor, f8, cortex_m_semihosting, 0x14, 0x28, 0x00, stm32f4, toabc, fnset, vec_closure

(Continued)

Table C.1. Continued

Topic (associated KU)	Word types
Aliasing	trait, type, parameter, define, alias, type_alias, defined, associated, definition, following, specify, named, defining, constrained, defines, rh, mul, inherent, unconstrained, ranked
Casting	convert, cast, casting, fromstr, type_inference, tryfrom, tostring, tryinto, type_conversion, conversion, converting, converted, try_from, try_into, tryfromsliceerror, i8, u16, digit, e0605, convertible
Control flow	loop, break, continue, condition, decision, control_flow, iteration, event, controlflow, looping, infinite, skip, std_iter, quit, subsequent, continuously, met, panic_handler, iterative, panicinfo
Distribution channels	nightly, stable, unstable_feature, night, nightlies, feature_flag, feature, rust_release, nightly_release, beta, build_run, experimental, stabilized, rust1, multirust, stability, intrinsic, gate, enabling, stabilization
Error handling	result, error, err, option, panic, panicking, abort, exception, error_handling, unwind, return, catch_unwind, std, unwrap, src, message, failure, invalid, success, fails
File I/O	std_io, file_read, file_descriptor, io, readonly, line_file, stdin, read, read_line, file_write, bufreader, buffer, bufread, reader, byte, expect, buf, stdout, input, line
Foreign function interface	extern, wrap, ffi, libc, no_mangle, c_void, bindgen, macro_use, stdcall, c_double, foreign_function, link_name, c_int, serde_derive, snappy, typedef, no_std, rustc_serialize, c_str, winapi
Functional language features	lifetime, parameter, closure, outlive, function, anonymous, infer, argument, fnmut, conflicting, anonymous_type, body, fnonce, functional_programming, anonymous_function, captured, capture, iterator_trait, callback, inferred
Lifetime	lifetime, reference, dropped, drop, scope, variable, block, live, lifetime_annotation, long, raii, valid, life, shadow, e0597, static, e0495, temporary, explicit, variable_binding
Macros	macro, procedural, procedural_macro, macro_rules, derive_macro, ident, expr, declarative, declarative_macro, tt, metaprogramming, expansion, ty, proc, originates, token, tokenstream, expand, generate, macro_export
Mutability	borrow, reference, mutable, mut, immutable, mutate, mutability, checker, borrows, occurs, mutably, refcell, ref, cell, e0502, borrow_mut, e0499, modify, mutation, content
Object Orientation	trait, implement, generic, method, implementing, object, implemented, inheritance, implementation, oop, composition, box, oo, boxed, object_oriented, polymorphism, partialord, ord, equality, delegate
Operators and symbols	function, type, const, array, expression, operator, constant, literal, signature, expected, i32, returning, mismatched, e0308, syntax, declare, opaque, expects, overloading, overload
Ownership	reference, borrow, mutable, ownership, value, owned, borrowing, pointer, moved, heap, copy, closure, stack, drop, borrowed, forget, clone, allocated, moving, owns
Patterns and Matchings	match, ok, err, let, pattern_matching, pattern_match, str, parse, matching, false, errorkind, from_str, eprintln, arm, msg, as_str, unreachable, matched, branch, parsing
Traits	trait, implement_trait, associated_type, trait_implementation, e0277, supertrait, satisfied, dyn, marker, subtrait, dispatch, impls, blanket, mytrait, e0599, unsized, vtable, e0119, implementors, default_type_parameter
Unsafe Rust	unsafe, safe, transmute, raw_pointer, unsafety, from_raw_parts_mut, ptr, memory_unsafety, mem, assume_init, soundly, as_ptr, safety, raw, undefined, as_mut_ptr, ub, repr, c_char, from_raw_parts
Async	future, async, tokio, mio, await, async_std, stream, poll, executor, map_err, ready, asynchronous, and_then, block_on, pin, reactor, fut, streamext, tokio_core, polled
Audio	think, thing, make, really, ha, like, good, time, sure, need, audio, know, bit, point, way, probably, better, look, used, problem
Command line arguments parsing	clap, subcommand, structopt, subcommands, from_os_str, argmatches, rdi, rax, mov, appsettings, rsi, rsp, eax, required_unless, rcx, rdx, load_yaml, retq, movq, ret
Command line interface	terminal, command_line, target, file, linux, run, command, running, ansi, bin, window, directory, linker, path, program, exit, failed, link, linking, script

(Continued)

Table C.1. Continued

Topic (associated KU)	Word types
Concurrency	thread, spawn, channel, mutex, std_sync, blocking, std_thread, task, threaded, process, concurrent, rayon, spawning, concurrently, concurrency, multithreaded, deadlock, mutexes, producer, send
Database	database, db, postgres, driver, sql, mysql, sqlite, postgresql, mongodb, sqlx, orm, elasticsearch, query, table, conn, id, schema, select, queryable, row
Embedded development	embedded, embed, interrupt, performance, faster, optimization, stm32, microcontroller, qemu, fast, usb, cpu, number, size, speed, slower, large, slow, small, firmware
Formatted print	println, print, format, main, to_string, output, printing, printed, formatting, push_str, formatted, num, formatting_trait, format_print, gen_range, format_trait, letter, printer, 02x, uncomment
Game development	game, rendering, physic, ggez, windowing, game_engine, draw, image, width, height, render, graphic, pixel, texture, opengl, gaming, roguelike, color, player, screen
Logging	log, logging, env_logger, levelfilter, log4rs, set_logger, syslog, simple_logger, loglevelfilter, setloggererror, fern, logger, console, appenders, production, rust_log, essential, pipeline, infrastructure, rolling
Network development	server, network, tcp, ip, openssl, std_net, udp, certificate, encryption, connection, socket, dns, tcpstream, ftp, net, bind, telnet, connect, smtp, tcplistener
Web development	http, server, web, html, hyper, actix, reqwest, diesel, rocket, actix_web, webassembly, http_request, request, cookie, client, cs, com, response, org, lang

D EXTRACTED QUOTES

In the following, we highlight the selected quotes from the three sources of information that we qualitatively analyzed to validate our results (see Table 7 in Section 5.2).

Structs:

[Q1] "*Define 'anonymous struct' before using it in section 6.1 (...) I cannot find a definition in sections 5 to 5.3, which introduce structs. I suggest either adding an explanation of anonymous structs to one of the structs sections (...)* "

https://github.com/rust-lang/book/issues/2584

[Q2] "*(...) example with Struct Update Syntax is too surface level and needs more in depth explanation (esp for new rustaceans). There's actually a lot more going on in the Struct Update Syntax example that's not explicitly explained or mentioned.* "

https://github.com/rust-lang/book/issues/2789

[Q3] "*Struct update syntax example doesn't make it clear that order doesn't matter* "

https://github.com/rust-lang/book/issues/1996

[Q4] "*Ambiguity regarding the possibility to use struct after move (...) Suggested fix: Rewrite the paragraph describing the possibility to use the whole struct after its field has been moved.* "

https://github.com/rust-lang/book/issues/2988

[Q5] "*I found the Listings on this section a bit confusing to follow because it is unclear on which scope the struct is being defined. (...) Suggested fix: A small sentence explaining in which scope the struct definition should go (...)* "

https://github.com/rust-lang/book/issues/2830

[Q6] "*The current page (...) doesn't really say much about unit structs. I think it could use some clarification.* "

https://github.com/rust-lang/reference/issues/1097

[Q7] "*Page on structs implies that unit struct definition is the same as struct struct definition + named constant* "

https://github.com/rust-lang/reference/issues/971

Patterns and Matchings:

[Q8] "*There is currently a section on destructuring pattern matching in Chapter 18.3. Would it be worth adding a paragraph, or a few sentences, mentioning destructuring assignment?* "

https://github.com/rust-lang/book/issues/2956

[Q9] "*Match guard example makes an incorrect statement (...) Suggested fix: Change the example to even/odd, so it's truly an example of something that can't be checked using a pattern.* "

https://github.com/rust-lang/book/issues/2868

[Q10] "*(...) a lot of emphasis is put in the fact that the compiler checks if match arms are exhaustive at compile time (...) What I think should also be mentioned is that it's not possible for the compiler to check if arms that contain expressions and conditionals are exhaustive or not (...) Suggested fix: Add an explanation of how exhaustive checking work with conditionals.* "

https://github.com/rust-lang/book/issues/2925

[Q11] "*Reference destructuring is mentioned but not taught (...) So what's the 'bug' in the book? I guess it should either say that reference destructuring is an advanced topic that isn't given as an example, or there should be examples for it?* "

https://github.com/rust-lang/book/issues/2733

[Q12] "*Add migration details to or-patterns guide (...)* "

https://github.com/rust-lang/edition-guide/pull/252

Linters:

[Q13] "*Document lint control scoping (...) the docs should be explicit how this scoping works (it's fairly unusual attribute behavior).* "

https://github.com/rust-lang/reference/issues/574

[Q14] "*Most of the 2021 lints are pretty obscure or complicated, and I think readers will get the gist of what it is saying.* "

https://github.com/rust-lang/edition-guide/pull/255

[Q15] "*Add rustdoc to tool lints* "

https://github.com/rust-lang/reference/pull/997

[Q16] "*Update lint level documentation (...) This adds some clarification about some details of lint level behavior.* "

https://github.com/rust-lang/reference/pull/998

[Q17] "*Describe the difference of* `rustc_lint` *vs* `rustc_lint_defs`. "

https://github.com/rust-lang/rustc-dev-guide/pull/1131

[Q18] "*Update information on lints (...) Clarify some information around sources of lints (...) Update information on future-incompatible lints (...)* "

https://github.com/rust-lang/rustc-dev-guide/pull/1154

[Q19] "*Document how lints are actually emitted (...) The section on linting describes how to declare a lint and how lint passes are constructed, but we are still missing an explanation on how the lint system takes buffered lints and decides whether to actually emit them to users.* "

https://github.com/rust-lang/rustc-dev-guide/issues/1168

Build:

[Q20] "*(...) the documentation, to which obvious effort has been devoted, is just not nearly good enough. For example, it is just about impossible to find how to get Cargo to link a given native library at a certain location. In other words, how to control the generated ld -l and -L options.* "

https://users.rust-lang.org/t/a-year-in-rust-and-i-still-cant-read-the-docs/46872

[Q21] "*Mention* `cargo test -- --include-ignored` *in test chapter (...) I was disappointed that the book didn't mention a way to run* `all` *tests, not just the* `ignored`. "

https://github.com/rust-lang/book/pull/2958

[Q22] "*Reflect the addition of the* `-p` *flag in Cargo 1.56 (...)* "

https://github.com/rust-lang/book/pull/3051

[Q23] "*The 'Getting Started' chapter is, TBH, pretty bad when it comes to the stuff about building and testing.* "

https://github.com/rust-lang/rustc-dev-guide/pull/1286

[Q24] "*– Add chapter on libs and metadata (...) This leans a bit heavy from a Cargo/build-tools perspective, but I think this information is useful. – I've wanted to see info about metadata and rustlibs in the dev-guide for a while, so thank you! – This is super helpful, thank you!* "

https://github.com/rust-lang/rustc-dev-guide/pull/1044

[Q25] "*(...) support for publishing a package from a Cargo workspace was added to the* `cargo publish` *command. The book however still mentioned that it wasn't possible to publish a package unless you were navigated into the respective package and published from there.* "

https://github.com/rust-lang/book/pull/3051

[Q26] "*Add* `cargo new` *learning to recap in ch01-03-hello-cargo.md. Although* `cargo new` *was explained in the previous subsection, it is altogether a very detailed section and it is good to include this for completeness and to remind the reader how to create a project, without them having to go back.* "

https://github.com/rust-lang/book/pull/3067

[Q27] "*Add [env] section to Cargo Reference - Configuration (...) This information is missing in the Cargo Reference/Configuration section.* "

https://github.com/rust-lang/book/issues/2665

[Q28] "*(...) as of Cargo 1.53 the author field is no longer added to the Cargo.toml (...) I just spent a day as a beginner reading the Cargo book and searching the web for ways to get it populated to my liking and eventually found a note to the above effect on Stack Overflow.* "

https://github.com/rust-lang/book/issues/2795

Foreign Function Interface:

[Q29] "*– (...) on 2018-edition book, I couldn't find any info about FFI (...) – There's no fantastic FFI documentation provided by the project right now.* "

https://users.rust-lang.org/t/where-can-i-get-the-latest-official-ffi-documents/21982

[Q30] "*linkage.md: update link to FFI section of the Book.* "

https://github.com/rust-lang/reference/pull/917

[Q31] "*Write a basic 'call Rust from C' example (...) this chapter seems to be getting quite long. At some point it might be worthwhile to consider breaking it up.* "

https://github.com/rust-lang/nomicon/pull/296

[Q32] "*ffi: explicitly declare* `hello_from_rust` *for C99* "

https://github.com/rust-lang/nomicon/pull/343

[Q33] "*On the FFI page, the Rust signatures of the functions* `snappy_compress`, `snappy_uncompress`, `snappy_uncompressed_length` *and* `snappy_validate_compressed_buffer` *are declared as returning* `c_int` *(...) Given that C does not guarantee that an enum's underlying type is* `int`*, shouldn't the return type on the Rust side be an enum declared with* `#[repr(C)]`*?* "

https://github.com/rust-lang/nomicon/issues/258

[Q34] "*Include brief explainer of C patterns and idioms in the FFI section (...) Ideally, the Nomicon should be enough to teach you everything you need to know about authoring an FFI API in Rust, without you needing to learn C to the level of knowing the idioms. (...) Right now the FFI chapter tells you how to link to another library, but not really how to write your own.* "

https://github.com/rust-lang/nomicon/issues/300

Macros:

[Q35] "*Where is the official documentation (up-to-date version) on writing procedural macros? (...) the chapter about the macros is missing and the reference says it is very incomplete.* "

https://users.rust-lang.org/t/official-documentation-for-writing-procedural-macros/24954

[Q36] "*I didn't find any documentation in the book that stated that the* `proc-macro = true` *statement should be inside the [lib] section.* "

https://github.com/rust-lang/book/pull/2915

[Q37] "*I find the 'new' way to write those macros with the declaration in the Cargo.toml blends in well with the import of macro crates. (...) Please update the Book to reflect those changes and requirements.* "

https://github.com/rust-lang/book/issues/2579

[Q38] "*There are no examples of how to do repetition on the page 'Macros By Example.' Including a simple example of e.g. how the* `vec!` *macro uses repetition would increase clarity. As someone who is not familiar with* `macro_rules!`*, I find the section on repetition very difficult to follow.* "

https://github.com/rust-lang/reference/issues/1123

[Q39] "*Improve the documentation of macros-by-example metavariable names (...) Metavariable names are not necessarily identifiers: they may also be keywords, raw identifiers or _.* "

https://github.com/rust-lang/reference/pull/1130

[Q40] "`ident` *macro fragment should include raw keywords. The ident fragment is currently documented as matching* `IDENTIFIER_OR_KEYWORD`*. However, I think it should also include* `RAW_IDENTIFIER`*.* "

https://github.com/rust-lang/reference/issues/610

[Q41] "*Document compatibility between declarative and procedural macro tokens.* "

https://github.com/rust-lang/reference/pull/1169

[Q42] "*Standardize reference of 'Attribute Macro' (...) The Attributes section uses 'macro attributes' but links to a section calling them 'attribute macros.'* "

https://github.com/rust-lang/reference/pull/958

[Q43] "*added* `macro_rules` *to weak keywords.* "

https://github.com/rust-lang/reference/pull/1008

[Q44] "*The description of 'procedural macros' was never updated when function-like procedural macros were stabilized.* "

https://github.com/rust-lang/reference/pull/1048

[Q45] "*(...) Add more detail on procedural macros and custom derive. The section on procedural macros is lacking (especially with regards to custom derive). This should be fleshed out more.* "

https://github.com/rust-lang/rustc-dev-guide/issues/1165

[Q46] "*(...) it needs to be mentioned that an attribute macro replaces the 'thing' they annotate (lost a ton of time in this yesterday).* "

https://github.com/rust-lang/book/issues/2267

[Q47] "*Proposal to change example for proc_macros (...) I think we should do this example the other way around.* "

https://github.com/rust-lang/book/issues/1639

[Q48] "*v1.58 introduces captured identifiers in format strings, and yet it seems most of the examples in the book still use positional placeholders.* "

https://github.com/rust-lang/book/issues/3047

Network Development:

[Q49] "*I would like to write a simple server with hyper that can support HTTPS (...) but documentation is Unclear.* "

https://users.rust-lang.org/t/unclear-documentation-in-hyper-openssl-0-5-0-for-server/17491

Installation and Setup:

[Q50] "*Chapter 1.1 - no mention of running* `rustup toolchain install stable` *(...) I believe it would be worth at least mentioning it in the book, even if it's not the recommended option.* "

https://github.com/rust-lang/book/issues/2901

[Q51] "*I think it would also make sense to include a note about how to get* `rustup link` *to use a cargo compiled with* `./x.py build cargo`*, but I haven't yet figured out how to do that.* "

https://github.com/rust-lang/rustc-dev-guide/pull/1273

File I/O:

[Q52] "*I'm reading about:* `std::io::BufReader` *and as a first example I see:* `reader.read_line (&mut line)?;`*. Then I am trying to find that function read_line at the left of the documentation, but it is not listed there!!!* "

https://users.rust-lang.org/t/documentation-of-types-discoverability-problem/53274

[Q53] "*(...) today I spent time on* `std::io::BufReader` *documentation (...) but I do not understand the following point (...) Does the* `std::io::BufReader<R>` *documentation states that I should not use a* `BufReader`*?* "

https://users.rust-lang.org/t/bufreader-documentation-precision/37903

Advanced Functions and Closures:

[Q54] "*Chapter 13.1 Mention why we put parenthesis around* `self.calculation` *when calling* `(self.calculation)(arg)` *(...) Suggested fix: Put a paragraph or an aside before the next code listing that mentions that (...) we are trying to invoke the closure stored in the field.* "

https://github.com/rust-lang/book/issues/2985

[Q55] "*19.4 Cannot use function pointer as return type (...) Suggested fix: Rephrase description or get rid of the function pointer mention altogether.* "

https://github.com/rust-lang/book/issues/3055

[Q56] "*19.1 has an unsound example and does not explain it (...) Suggested fix: Either mention that it is unsound and this code should not be copied in prose, or change the example.* "

https://github.com/rust-lang/book/issues/3013

[Q57] "*Provide more information about disjoint capture in closures and migration instructions.* "

https://github.com/rust-lang/edition-guide/issues/241

[Q58] "*Chapter 19.4: impl vs box dyn (...) This section shows a compiler error suggesting using* `impl`*, then (without comment) gives a different solution using* `Box<dyn>` *(...) Either use* `impl` *(as the compiler suggests) or comment on the difference between the solutions.* "

https://github.com/rust-lang/book/issues/2872

[Q59] "*ch19-05: Enhance* `Returning Closures` *section (...) This PR enhances* `Returning Closures` *section from ch19-05. It mentions the* `impl Trait` *syntax for returning closures from functions, as another way except Trait objects.* "

https://github.com/rust-lang/book/pull/2725

[Q60] "*Clarify the HRTB chapter (...) Recently, someone asked for some clarifications in the _easy questions_ thread on the unofficial Rust subreddit regarding the HRTB chapter (...)* "

https://github.com/rust-lang/nomicon/pull/330

[Q61] "*Consolidate HKTBs (...) There's great information about all these concepts (...) Would be great to review them and add stuff into the guide.* "

https://github.com/rust-lang/rustc-dev-guide/issues/1184

Database Development:

[Q62] "*The diesel project is looking for feedback and additional input on their documentation for the upcoming 2.0 release.* "

https://users.rust-lang.org/t/a-year-in-rust-and-i-still-cant-read-the-docs/46872

[Q63] "*I'd love to see some 'database book' that covers connections, drivers, pools, has examples for every feature and gives overview of ecosystem. Individual project would benefit greatly from better docs.* "

https://internals.rust-lang.org/t/kickstarting-a-database-wg/9696/5

REFERENCES

[1] 2007. The Java SE Documentation Survey. Retrieved from https://blogs.oracle.com/thejavatutorials/the-java-se-documentation-survey.
[2] 2018. Where is "use crate:" documented? Retrieved from https://users.rust-lang.org/t/20910.
[3] 2019. Documentation (eg. of "-> !"). Retrieved from https://users.rust-lang.org/t/35764.
[4] 2020. Go Developer Survey 2020 Results. Retrieved from https://blog.golang.org/survey2020-results#TOC_9.
[5] 2020. Kickstarting a database-wg. Retrieved from https://internals.rust-lang.org/t/kickstarting-a-database-wg/9696/5/.
[6] 2020. Python Developers Survey 2020 Results. Retrieved from https://www.jetbrains.com/lp/python-developers-survey-2020/.
[7] 2020. Rust 2020 Survey Results. Retrieved from https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html.
[8] 2020. Rust Game Development - Ecosystem Survey. Retrieved from https://gamedev.rs/blog/survey-01/.
[9] 2020. Rust Survey 2019 reveals the need for more documentation. Retrieved from https://jaxenter.com/rust-survey-2019-171246.html.
[10] 2020. Stack Overflow Developer Survey 2020. Retrieved from https://insights.stackoverflow.com/survey/2020.
[11] 2020. User & Developer Survey 2020. Retrieved from https://julialang.org/assets/2020-julia-user-developer-survey.pdf.
[12] 2021. Anchored CorEx. Retrieved from https://github.com/gregversteeg/corex_topic/.
[13] 2021. The Cargo Book. Retrieved from https://doc.rust-lang.org/cargo/.
[14] 2021. Discourse. Retrieved from https://www.discourse.org/.
[15] 2021. Discourse API Documentation. Retrieved from https://docs.discourse.org/.
[16] 2021. The Edition Guide. Retrieved from https://doc.rust-lang.org/edition-guide/.
[17] 2021. The Embedded Rust Book. Retrieved from https://docs.rust-embedded.org/book/.
[18] 2021. Results of the First Kotlin Multiplatform Survey. Retrieved from https://blog.jetbrains.com/kotlin/2021/01/results-of-the-first-kotlin-multiplatform-survey/.
[19] 2021. Rust by example. Retrieved from https://doc.rust-lang.org/rust-by-example/.
[20] 2021. The Rust core allocation and collections library. Retrieved from https://doc.rust-lang.org/alloc/.
[21] 2021. The Rust Core Library. Retrieved from https://doc.rust-lang.org/core/.
[22] 2021. Rust official learning material. Retrieved from https://www.rust-lang.org/learn.
[23] 2021. The Rust Programming Language. Retrieved from https://doc.rust-lang.org/book/.
[24] 2021. The Rust Reference. Retrieved from https://doc.rust-lang.org/stable/reference/.
[25] 2021. The Rust Standard Library. Retrieved from https://doc.rust-lang.org/std/.
[26] 2021. The rustc Book. Retrieved from https://doc.rust-lang.org/rustc/.
[27] 2021. The rustdoc Book. Retrieved from https://doc.rust-lang.org/rustdoc/.
[28] 2021. The Rustonomicon. Retrieved from https://doc.rust-lang.org/nomicon/.
[29] 2021. Stack Exchange Data Explorer. Retrieved from https://data.stackexchange.com/.
[30] 2021. Supplementary material. Retrieved from https://www.dropbox.com/sh/bkeewtymjnf9yp6/AAAstyl4-r2FZCfWkzkBoyyma?dl=0.
[31] 2022. Annual Rust Survey design. Retrieved from https://github.com/rust-lang/surveys/blob/main/surveys/notes/design.md.
[32] 2022. Annual Rust Survey questions. Retrieved from https://github.com/rust-lang/surveys/blob/main/surveys/2021-annual-survey/questions.md.
[33] 2022. Discussion to kickstart the Rust Survey for 2020. Retrieved from https://github.com/rust-lang/core-team/issues/7#issuecomment-663220142.
[34] 2022. Documentation on the Survey Platform. Retrieved from https://github.com/rust-lang/surveys/blob/main/surveys/notes/survey-platforms.md.
[35] 2022. Ensure stakeholders have been consulted about the 2021 Survey Design. Retrieved from https://github.com/rust-lang/surveys/issues/3.
[36] 2022. Getting to a wider feedback round (Annual Rust Survey). Retrieved from https://github.com/rust-lang/surveys/issues/63.

[37] 2022. How can I convert it back into bson::document::Document on the server side? Retrieved from https://stackoverflow.com/questions/63074618.
[38] 2022. How to import all macros, derives, and procedural macros in Rust 2018 without using extern crate? Retrieved from https://users.rust-lang.org/t/33225.
[39] 2022. Intercepting libc readlink with a RUST LD_PRELOAD program hangs when applying to cargo build. Retrieved from https://users.rust-lang.org/t/48057.
[40] 2022. Is there an idiomatic way to implement the component pattern? Retrieved from https://stackoverflow.com/questions/31897986.
[41] 2022. Log4rs - how to use different levels for appenders. Retrieved from https://stackoverflow.com/questions/50999749.
[42] 2022. Rust Annual Survey Repository. Retrieved from https://github.com/rust-lang/surveys.
[43] 2022. Rust Annual Survey Tentative Timeline. Retrieved from https://github.com/rust-lang/surveys/tree/main/surveys/2021-annual-survey.
[44] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. 2020. Challenges in chatbot development: A study of Stack Overflow posts. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 174–185.
[45] Syed Ahmed and Mehdi Bagherzadeh. 2018. What do concurrency developers ask about? A large-scale study using Stack Overflow. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[46] Anwar Alqaimi, Patanamon Thongtanunam, and Christoph Treude. 2019. Automatically generating documentation for Lambda expressions in Java. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories*. 310–320.
[47] Moayad Alshangiti, Hitesh Sapkota, Pradeep K. Murukannaiah, Xumin Liu, and Qi Yu. 2019. Why is developing machine learning applications challenging? A study on Stack Overflow posts. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–11.
[48] Mohab Aly, Foutse Khomh, and Soumaya Yacout. 2021. What do practitioners discuss about IoT and industry 4.0 related technologies? Characterization and identification of IoT and Industry 4.0 categories in Stack Overflow discussions. *Internet Things* 14 (2021), 100364.
[49] David Andrzejewski, Xiaojin Zhu, and Mark Craven. 2009. Incorporating domain knowledge into topic modeling via Dirichlet forest priors. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 25–32.
[50] Richard A. Armstrong. 2014. When to use the Bonferroni correction. *Ophthalm. Physiolog. Optics* 34, 5 (2014), 502–508.
[51] S. Arora, R. Ge, and A. Moitra. 2012. Learning topic models—going beyond SVD. In *Proceedings of the IEEE 54th Annual Symposium on Foundations of Computer Science*. 1–10.
[52] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going big: A large-scale study on what big data developers ask. In *Proceedings of the ACM 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 432–442.
[53] Abdul Ali Bangash, Hareem Sahar, Shaiful Chowdhury, Alexander William Wong, Abram Hindle, and Karim Ali. 2019. What do developers know about machine learning: A study of ML discussions on Stack Overflow. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories*. 260–264.
[54] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. 2014. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empir. Softw. Eng.* 19 (2014), 619–654.
[55] David M. Blei and John D. Lafferty. 2006. Dynamic topic models. In *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*. 113–120.
[56] Raymond P. L. Buse and Westley R. Weimer. 2008. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis*. 273–282.
[57] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Gias Uddin. 2021. How do developers discuss and support new programming languages in technical Q&A site? An empirical study of Go, Swift, and Rust in Stack Overflow. *Inf. Softw. Technol.* 137 (2021), 106603.
[58] Chunyang Chen and Zhenchang Xing. 2016. Towards correlating search on Google and asking on Stack Overflow. 83–92.
[59] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762.
[60] Norman Cliff. 1996. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press, New York.
[61] Daniela Soares Cruzes and Tore Dybå. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. 275–284.

[62] Uri Dekel and James D. Herbsleb. 2009. Reading the documentation of invoked API functions in program comprehension. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*. 168–177.
[63] Ying Fang, Heyan Huang, Ping Jian, Xin Xin, and Chong Feng. 2014. Self-adaptive topic model: A solution to the problem of "rich topics get richer." *China Commun.* 11, 12 (2014), 35–43.
[64] Ryan Gallagher, Kyle Reing, David Kale, and Greg Ver Steeg. 2017. Anchored correlation explanation: Topic modeling with minimal domain knowledge. *Trans. Assoc. Computat. Ling.* 5, 0 (2017), 529–542.
[65] Aria Haghighi and Dan Klein. 2006. Prototype-driven learning for sequence models. In *Proceedings of the Human Language Technology Conference of the NAACL*. 320–327.
[66] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do programmers discuss about deep learning frameworks. *Empir. Softw. Eng.* 25 (2020), 2694–2747.
[67] Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. 2012. Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers? In *Proceedings of the 28th IEEE International Conference on Software Maintenance*. 243–252.
[68] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. 200–210.
[69] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empir. Softw. Eng.* 20 (2005), 31–55.
[70] Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*. 344–353.
[71] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* 32, 12 (2006), 971–987.
[72] L. I. Kuncheva and J. C. Bezdek. 1998. Nearest prototype classification: Clustering, genetic algorithms, or random search? *IEEE Trans. Syst. Man Cybern. Part C (Applic. Rev.)* 28, 1 (1998), 160–164. DOI: https://doi.org/10.1109/5326.661099
[73] T. C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE Softw.* 20, 6 (2003), 35–39. DOI: https://doi.org/10.1109/MS.2003.1241364
[74] Mario Linares-Vásquez, Bogdan Dit, and Denys Poshyvanyk. 2013. An exploratory analysis of mobile development issues using Stack Overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. 93–96.
[75] Paul W. McBurney, Siyuan Jiang, Marouane Kessentini, Nicholas A. Kraft, Ameer Armaly, Mohamed Wiem Mkaouer, and Collin McMillan. 2018. Towards prioritizing documentation effort. *IEEE Trans. Softw. Eng.* 44, 9 (2018), 897–913.
[76] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.
[77] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st International Conference on Program Comprehension*. 23–32.
[78] Michael Moser, Josef Pichler, Günther Fleck, and Michael Witlatschil. 2015. RbG: A documentation generator for scientific and engineering software. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 464–468.
[79] Allen Newell. 1963. Documentation of IPL-V. *Commun. ACM* 6, 3 (1963), 86–89.
[80] Akond Rahman, Asif Partho, Patrick Morrison, and Laurie Williams. 2018. What questions do programmers ask about configuration as code? In *Proceedings of the IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering*. 16–22.
[81] Louis M. Rea and Richard A. Parker. 2014. *Designing and Conducting Survey Research: A Comprehensive Guide* (4th ed.). Jossey-Bass.
[82] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-demand developer documentation. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 479–483.
[83] Michael Röder, Andreas Both, and Alexander Hinneburg. 2015. Exploring the space of topic coherence measures. In *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*. 399–408.
[84] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys? In *Proceedings of the Annual Meeting of the Florida Association of Institutional Research*. 1–3.
[85] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? A large scale study using Stack Overflow. *Empir. Softw. Eng.* 21 (2016), 1192–1223.
[86] Christopher J. Shaw. 1963. Jovial and its documentation. *Commun. ACM* 6, 3 (1963), 89–91.

[87] Lucas B. L. Souza, Eduardo C. Campos, Fernanda Madeiral, Klerisson Paixao, Adriano M. Rocha, and Marcelo de Almeida Maia. 2019. Bootstrapping cookbooks for APIs from crowd knowledge on Stack Overflow. *Inf. Softw. Technol.* 111 (2019), 37–49.

[88] Greg Ver Steeg and Aram Galstyan. 2014. Discovering structure in high-dimensional data through correlation explanation. In *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems*. 577–585.

[89] Florian Steuber, Mirco Schoenfeld, and Gabi Dreo Rodosek. 2020. Topic modeling of short texts using anchor words. In *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*. 210–219.

[90] E. Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the IEEE/ACM 2nd International Conference on Software Engineering*. 492–497.

[91] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web? In *Proceedings of the IEEE/ACM 33rd International Conference on Software Engineering*. 804–807.

[92] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. 2015. Extracting development tasks to navigate software documentation. *IEEE Trans. Softw. Eng.* 41, 6 (2015), 565–581. DOI: https://doi.org/10.1109/TSE.2014.2387172

[93] Pradeep K. Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed E. Hassan. 2016. What do client developers concern when using web APIs? An empirical study on developer forums and Stack Overflow. In *Proceedings of the IEEE International Conference on Web Services*. 131–138.

[94] Zhiyuan Wan, Xin Xia, and Ahmed E. Hassan. 2019. What is discussed about blockchain? A case study on the use of balanced LDA and the reference architecture of a domain to capture online discussions about blockchain platforms across the stack exchange communities. *IEEE Trans. Softw. Eng.* (2019), 1–19.

[95] Shaowei Wang, David Lo, and Lingxiao Jiang. 2013. An empirical study on developer interactions in Stack Overflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1019–1024.

[96] David Weltman and Mark Eakin. 2018. Basic guidelines for common business statistics metrics. *Bus. Educ. Innov. J.* 10, 2 (2018), 21–26.

[97] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from Stack Overflow? *Empir. Softw. Eng.* 24, 2 (2019), 637–673.

[98] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What security questions do developers ask? A large-scale study of Stack Overflow posts. *J. Comput. Sci. Technol.* 31 (2016), 910–924.

[99] Victor H. Yngve and Jean E. Sammet. 1963. Toward better documentation of programming languages. *Commun. ACM* 6, 3 (1963), 76.

[100] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*. 380–391.

Received 27 October 2021; revised 8 May 2022; accepted 11 June 2022