Evaluating Software Documentation Quality

Henry Tang, Sarah Nadi
University of Alberta
{hktang, nadi}@ualberta.ca

***Abstract*—The documentation of software libraries is an essential resource for learning how to use the library. Bad documentation may demotivate a developer from using the library or may result in incorrect usage of the library. Therefore, as developers select which libraries to use and learn, it would be beneficial to know the quality of the available documentation. In this paper, we follow a systematic process to create an automatic documentation quality evaluation tool. We identify several documentation quality aspects from the literature and design metrics that measure these aspects. We design a documentation quality overview visualization to visualize and present these metrics, and receive intermediate feedback through a focused interview study. Based on the received feedback, we implement a prototype for a web service that can evaluate a given documentation page for Java, JavaScript, and Python libraries. We use this web service to conduct a survey with 26 developers where we evaluate the usefulness of our metrics as well as whether they reflect developers' experiences when using this library. Our results show that participants rated most of our metrics highly, with Text Readability, and Code Readability (of examples) receiving the highest ratings. We also found several libraries where our evaluation reflected developers' experiences using the library, indicating the accuracy of our metrics.**

I. Introduction

Software libraries provide reusable code that helps developers quickly integrate specific functionality. To benefit from such code reuse, developers usually first compare multiple libraries that offer the needed functionality [1]–[3] and then spend time to learn the selected library for efficient integration into their code [4]. Fortunately, most libraries include documentation that helps in library selection [2], [5]–[11] as well as in learning how to use a library. However, if the documentation is low quality then it is not only ineffective in assisting developers to use the library, it may cause further confusion, e.g., if the information is incorrect or out of date. Libraries with higher documentation quality will take less time for a developer to learn, which raises the question of "*What are indicators of documentation quality?*" In fact, in previous research on selecting and comparing software libraries across different metrics, the most demanded additional metric to include was the quality of the library's documentation [2].

While there are existing efforts for evaluating and understanding documentation quality, they either focus on a few characteristics that may not be relevant to library selection [12], [13], such as documentation page clones [13], or do not provide accessible automated implementations for measuring these characteristics [14]–[16]. In this paper, our goal is to create an automatic documentation quality evaluation tool to aid developers in comparing and selecting a library to learn and use.

We begin with an overview of related work on software documentation (Section II), which we use to find documentation quality aspects. We prioritize these aspects according to their importance in the literature and focus on five aspects: completeness, readability, code examples, ease of use (of documentation), and up-to-dateness. We then use the Goal-Question-Metric (GQM) paradigm [17] to define nine metrics to measure theses aspects. We create an initial mockup of our documentation quality summary based on these metrics (shown in Figure 1) and conduct an interview with three professional developers for early feedback (Section IV). Based on the feedback we receive, we implement and verify all metrics and create a public web application prototype (Figure 2) that allows developers to visualize the documentation quality summary of any Python, Java, or Javascript library (Section III). Finally, we use this web application to conduct a broader survey evaluation of our metrics and documentation quality summary page.

We received 26 survey responses for 12 of 40 analyzed libraries across the 3 supported languages. Overall, participants found our summary to be useful (median score 4/5). We find that the most highly rated metrics are the "Text readability" and "Code readability" metrics, suggesting that library maintainers should focus on clearly communicating information through the descriptions and code examples in the documentation. Our work not only provides developers a tool to quickly evaluate a library documentation's quality, but it also assists library maintainers by highlighting the documentation quality aspects they should focus on. Our work also opens up the possibility of creating additional metrics to measure other documentation aspects, as well as extending our current prototype for additional programming languages.

The contributions of this paper are as follows.

- The definition, automation, and verification of nine metrics for evaluating documentation quality.
- An interview study with three professional developers, and a survey evaluation with 26 software developers to verify our metrics and presentation.
- A prototype for a web application [1] that accepts a library link as input and produces an overview summary of a library's documentation quality. To date, the service processed 40 libraries (22 Python libraries, 5 Java libraries, and 13 Javascript libraries).

Our artifact page [18] contains our prototype implementation and data from 40 libraries.

[1]https://smr.cs.ualberta.ca/docquality/

II. Background & Literature Review

A. Types of Documentation & Scope

Software documentation can describe various types of documents that appear during the software development and maintenance lifecycle [5], [19], ranging from requirements, architectural documents, to source code comments. Documentation can also have different target audiences, e.g., the system's developers, maintainers, or end users. In our work, we focus on official documentation for a library's client developers [11], [20]–[22], which typically conveys information about the public Application Programming Interface (API), installation instructions, or other information on using the library.

B. Documentation Quality Aspects

To develop metrics that evaluate library documentation quality, we first need to identify which aspects affect documentation quality. Table I summarizes our study of previous research; it shows each documentation quality aspect mentioned in the literature, its definition, and the references that identify/consider this aspect.

Forward and Lethbridge [5] conduct a survey to understand the kinds of documentation used by various roles in a software project (e.g., architects, developers, managers) and which documentation attributes are important across these roles. Robillard [7] surveyed 80 participants to explore the issues developers face when learning APIs, and Uddin and Robillard [25] surveyed 69 developers to collect common documentation problems, which they combine into 10 categories (referenced in Table I). All of these efforts revealed similar documentation issues including, but not limited to: insufficient or inadequate examples, incomplete content, and no reference on accomplishing specific tasks with the API.

While there is additional work that also explores the issues and attributes of documentation [4], [19], [23], [24], [27], [29], [30], they include the same documentation quality aspects discussed above. Table I summarizes all quality aspects (ordered alphabetically) and the references that discuss them in. In Section III, we use this table to select the documentation aspects to measure.

C. Measuring Documentation Quality

We now discuss previous efforts in automatically evaluating documentation quality. Forward [12] explored the creation of a documentation quality indicator by combining metrics of an **A**rtefact's **U**sefulness, **R**eferential decay, and **A**uthority, or AURA. Forward [12] defined *usefulness* as a measure of a document's recency, frequency, and feedback, which requires long term monitoring of a document. *Referential decay* uses a document's last modified date to calculate its relative inconsistency with the source code. Finally, Forward [12] uses Kleinberg's authoritive ranking technique [31] as a measure of a document's authority, indicating the importance of a document compared to other documents with similar information (e.g., external blogs). Although these metrics are potentially useful, AURA requires long term data collection in order to calculate its metrics.

TABLE I: Summary of Documentation Quality Aspects

Aspect	Definition	References
Accessibility	How difficult is it to find the documentation?	[5], [9], [10]
Appeal	How interesting is it to read?	[23]
Appropriateness comments	Density of source code comments	[14]
Authority	How much authority does the documentation have?	[5], [9], [12]
Code examples	The existence of code examples in the documentation	[4], [5], [7], [8], [24], [25]
Cohesion	How well does the documentation fit together?	[23]
Completeness	Is the information in the documentation complete? Is all the source code documented? Is all the tasks/features documented?	[7]–[10], [14], [15], [25]–[28]
Consistency	Is the documentation consistent with itself? Same terminology, same format, etc.	[8], [9], [22], [23], [25], [26]
Consistency (to standard)	Does the documentation conform to a documentation standard defined by an external authority?	[10], [14]
Correctness	Is the information in the documentation accurate?	[8], [9], [25], [27]
Documentation PoV	The documentation should be written in the point of view of the reader.	[29]
Ease of Use	How easy is the documentation able to be used? e.g., navigation, internal and external links	[4], [5], [14], [24], [25]
Effectiveness	Does the documentation make effective use of technical vocabulary?	[23]
Fitness of purpose	Does the documentation fit its intended purpose	[29]
Format	What file format is the documentation? e.g., HTML, PDF, etc.	[5]
Graphical Support	Does the documentation use images?	[5], [8], [14]
Length	The length of the sentences in the documentation.	[5], [14]
Maintainability	How easy is the documentation able to be updated?	[27]
Organization	Is the information in the documentation organized efficiently? e.g., Sections/subsections	[5], [7]–[10], [14], [23], [25], [29]
Preciseness	How precise is the documentation?	[8]
Quality	How well written is the documentation?	[23]
Readability	How easy is the documentation read?	[5], [8], [9], [14], [22], [23], [25], [27]
Record rationale	Does the documentation include design decisions?	[29]
Relevance of content	Is the information in the documentation relevant?	[5], [7], [8], [23], [25], [29]
Spelling and Grammar	Spelling and grammar of the documentation	[5], [9]
Support many scenarios	Does the documentation support many scenarios?	[7]
Traceability	What is the extent to which changes in the documentation can be tracked?	[9]
Type	What is the type of documentation? e.g., Requirements, Specifications, Testing, etc.	[5]
Understandability	How understandable is the information in the documentation?	[23], [25], [29]
Up-to-date	How up to date is the documentation relative to the source code?	[5], [8]–[10], [12], [14], [22], [25]–[27], [29]
Usability	To what degree in which users can use the documentation to achieve objectives?	[27]
Usefulness	How useful is the documentation?	[12]

Aversano et al. [14] evaluated the quality of documentation of open-source software. In their work, they define multiple aspects of documentation quality, such as: completeness, alignment (i.e., up-to-dateness), readability, and ease of use. Many of these aspects were also identified by previous researchers (Section II-B). Although Aversano et al. [14] do define automatically measurable metrics for aspects of documentation, they do not actually implement these metrics in a usable tool or evaluate if these metrics adequately measure the intended aspects. Note that our completeness and readability metrics match those used by Aversano et al. [14].

Other attempts at evaluating parts of software documentation include Zhong and Su's [15] DocRef tool and Lee et al.'s [16] FreshDoc tool, both of which attempt to find

TABLE II: Summarized GQM for the five selected aspects

Goal/Aspect	Question(s)	Metric(s)
Up-to-date	Is the documentation consistent with the source code?	**(M1)** Ratio of matching public methods between documentation and source code **(M2)** Ratio of matching public classes between documentation and source code
Completeness	- Are the methods/classes found in the documentation in the source code? - What are the documented tasks and/or features ?	**(M1)** Ratio of matching public methods between documentation and source code **(M2)** Ratio of matching public classes between documentation and source code **(M3)** Documented task list
Readability	- Is the documentation text readable? - Are the code examples readable?	**(M4)** Flesch readability score [32] **(M5)** Code readability metric from previous literature [33]–[35]
Code examples	Do all public API methods/classes have code examples?	**(M6)** Ratio of public API methods with code examples **(M7)** Ratio of public API classes with code examples
Ease of use (of documentation)	How easy is it to navigate the documentation?	**(M8)** HCI web page navigation checklist [36]–[38] **(M9)** Quick start check

discrepancies between the documentation and source code.

Our work is different from the above in that we provide (1) a comprehensive summary of documentation quality that addresses multiple important documentation aspects, (2) an automatic service that evaluates documentation quality, and (3) validated metrics through manually constructed ground truths, developer interviews, and survey participants.

III. Selection of Aspects and Metrics

We use our literature review from Section II to guide our selection of aspects to develop metrics for. We do not discuss concrete metric implementations here, but rather in Section V.

a) Selecting documentation aspects: To avoid overwhelming developers with too much data about all possible metrics for evaluating documentation quality, we focus on a subset of documentation aspects as follows. First, we look at the number of supporting literature references of an aspect. Since many of these references base their findings on developer input, aspects mentioned repeatedly suggest that they are more important to developers in assessing documentation. However, sometimes these aspects are difficult to objectively *and automatically* measure. For example, the "Organization" aspect in Table I has relatively many supporting references, but is difficult to measure as there is no standard for what is considered "organized". Therefore, we also consider whether aspects are "potentially measurable" (i.e., can be objectively and automatically calculated) as an additional selection criteria.

We use Aghajani et al.'s [19], [27] comprehensive taxonomy of documentation issues and Forward and Lethbridge's [5] attribute list as a starting point for aspects to focus on. These two sources help us narrow our focus on documentation quality aspects that developers find hindering their learning of a library, corresponding to our goal of focusing on the end users of a library. From these two sources, we select the following measurable aspects: Completeness, Readability, and Up-to-dateness. From this starting point, we then look at the other documentation aspects from the literature in Table I and additionally select the next most supported and measurable aspects, which are Code examples and Ease of Use. Our final selection of documentation aspects to measure are: (1) Up-to-date, (2) Completeness, (3) Readability, (4) Code examples, and (5) Ease of Use (of documentation).

b) Deriving Metrics for Selected Documentation Aspects: The Goal-Question-Metric (GQM) paradigm [17] is an approach that starts from a high-level "goal", where the idea is to ask "questions" about the goal, and ends with "metrics" that answer the "questions" and measure the "goal". For example, the "Code examples" aspect is about the existence of code examples in the documentation, which we can turn into a "goal", i.e., the goal is to have code examples in the documentation. Then, we move on to ask "questions", where the purpose of the "questions" is to explore and define objects that represent the goal. One question a developer may ask for the "Code examples" aspect might be *"Do all public API methods/classes have code examples?"* Finally, "metrics" are created to measure answers for these questions, e.g., measure how many code examples exist in the documentation for a method or class of the library. Table II summarizes our GQM [17] process for the five selected aspects. We discuss the details of each metric and how we implement its measurement in Section V. Note that some metrics can measure multiple goals (aspects), while multiple metrics may be needed to measure one goal (aspect). Overall, we derive eight metrics at this point of our work (numbered M1 to M8); we added the ninth metric (M9) later based on our interview study, described next.

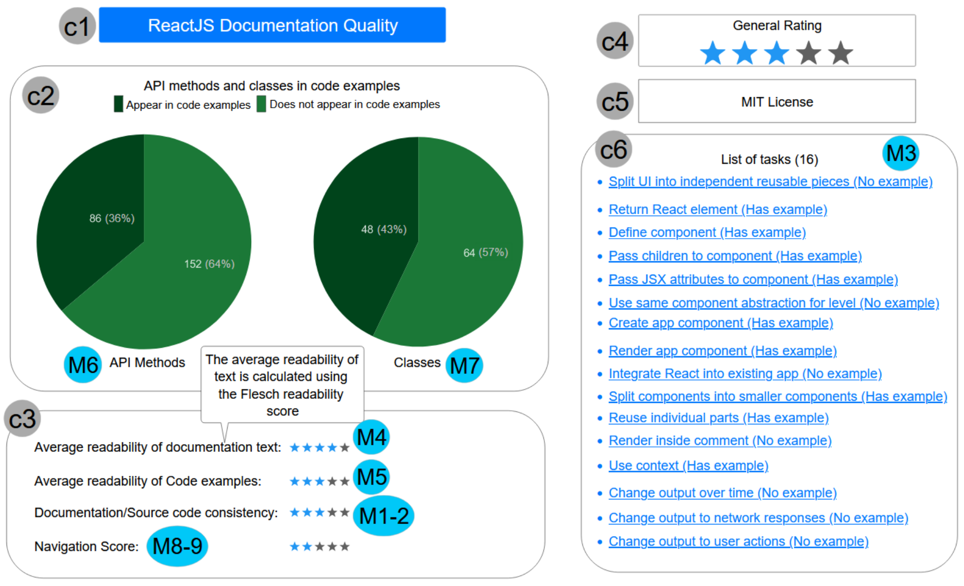
IV. Initial Interview Study

Before fully implementing the metrics, we create a mockup (Figure 1) of how we want to present and summarize the metric information. To ensure that we build tooling that is useful to developers, we conduct an intermediate small interview study as a checkpoint. Our goal is to get early feedback regarding the following questions: (1) Are our metrics useful in evaluating documentation quality?, (2) Do our metrics measure the documentation aspects they intend to measure? and (3) Is our presentation of each metric easy-to-read and understand?

A. Interview Study Setup

1) Presented Mockup: To provide a realistic mockup with reasonable values, we manually calculated the values of the metrics for one page of the ReactJS documentation [39] and used this mockup to receive intermediate feedback from developers through our interview study.

Our initial summary visualization shown in Figure 1 has six components, labelled as c1 - c6. Although components c1 (name) and c5 (license) are not any of our metrics, we include them to provide background information about the library. Component c2 is a visualization of the individual metrics measuring the number of public API methods/classes that have a code example provided in the documentation (metrics M6 and M7 from Table II). Component c3 displays multiple metrics. The "Average readability of x" refers to the rating of either the documentation text readability or the code example readability (metrics M4 and M5). Component c3 also includes the two ratios of matching methods/classes


c1
ReactJS Documentation Quality
c2
API methods and classes in code examples
Appear in code examples
Does not appear in code examples
86 (36%)
152 (64%)
48 (43%)
64 (57%)
M6
API Methods
Classes
M7
The average readability of text is calculated using the Flesch readability score
c3
M4
M5
M1-2
M8-9
c4
General Rating
★★★★★
c5
MIT License
c6
M3

Average readability of documentation text: ★★★★★

Average readability of Code examples: ★★★★★

Documentation/Source code consistency: ★★★★★

Navigation Score: ★★★★★

List of tasks (16)

- Split UI into independent reusable pieces (No example)
- Return React element (Has example)
- Define component (Has example)
- Pass children to component (Has example)
- Pass JSX attributes to component (Has example)
- Use same component abstraction for level (No example)
- Create app component (Has example)
- Render app component (Has example)
- Integrate React into existing app (No example)
- Split components into smaller components (Has example)
- Reuse individual parts (Has example)
- Render inside comment (No example)
- Use context (Has example)
- Change output over time (No example)
- Change output to network responses (No example)
- Change output to user actions (No example)

Fig. 1: Initial mockup of documentation quality overview. We validate this mockup in our interview study.

in the documentation with the source code (metrics M1 and M2) displayed as one metric labelled "Documentation/Source code consistency". We do not use the term "up-to-date" in our visualization because "up-to-date" implies accompanying information (e.g., textual descriptions, code examples, etc.), about the API is also "up-to-date" [16], [40]; however, we only measure matching signatures and definitions which we combine through an equally weighted sum. The last metric presented in Component c3 is the "Navigation Score" (metric M8). Component c6 is the "List of tasks"(metric M3), which details what tasks can be done with the library, as extracted from the documentation. It also provides information on whether each task has an accompanying code example in the documentation between parentheses. Finally, component c4 is a "general rating" of the documentation quality, which is an equal weighted combination of the previous metrics.

2) Participants and Interview Structure: Our participants are professional software developers, working in an industrial research company, and developing mainly using Javascript. After receiving ethics clearance from our university, we interview three developers of varying seniority and experience from this company. P1 is a junior R&D Engineer with 4 years experience, P2 is a junior developer with 2 years experience, and P3 is a senior R&D Engineer with 12 years experience. We conduct the interviews online through Zoom following a semi-structured setup [41], where we use pre-determined guiding questions but allow the discussion to deviate and explore tangent directions. We kick-start the discussion with open-ended questions about what they perceive as positive and negative features in library documentation (i.e., what do they like seeing in documentation and what do they dislike seeing). We then show participants our documentation quality mockup, Figure 1. For each component in the mockup, we ask developers whether (1) the metric adequately represents its intended aspect and (2) whether that aspect is beneficial in evaluating documentation quality. When a participant did not understand the goal or intention of a metric, we noted that and explained it to them before asking them again to answer these two questions. Each interview lasted $\sim$ 30 minutes and was recorded (with participant consent and ethics clearance) for later transcription and analysis.

B. Interview Results

Due to space limitations, we focus on presenting the results related to our documentation quality summary. Our artifact [18] includes additional discussions from the interviews.

1) General mockup feedback: Generally, all participants found our mockup clean and appreciated the current information presented as an overall report on a library's documentation. However, they had several comments about presentation such as using the space more effectively to draw more attention to important information. For example, component c2, though important, takes too much space on the page, and instead, that space could be used to make the "List of Tasks" (component c6) more prevalent (which all participants found important). As such, participants suggested changing the structure of the different components, i.e., placing more important information (e.g., list of tasks, general rating) on the left side, and moving the metrics to the right side. Participants also suggested using more succinct labels for some metrics while at the same time better conveying what the metric is trying to measure, e.g., changing "Average readability of documentation text" to "Readability of text".

2) API methods/classes in Code Examples (c2): We got a unanimous "yes" about the usefulness of this component, with P3 responding, *"'...hopefully I would like to have 100 percent every time..."*. However, participants suggested replacing the (unnecessary) pie charts with a single line for each metric.

3) Documentation & code example readability c3: All participants responded that readability of text and code examples are important. Participants did not want to waste time or effort understanding complex explanations and prefered short, concise sentences that convey the intent of what is being described. Participants also think that readability of code examples is important. Since developers gravitate toward code examples to quickly understand how to use the library, its clarity and understandability is necessary for developers to feel comfortable with the library enough to experiment with it. All participants conveyed their satisfaction with the way we presented this information in the mockup, with P2 stating *"You give me one important information, because average readability of documentation is something I value a lot. ...[Y]ou gave me that is four out of five and you gave me information on how you calculate it. So it's perfect."*

4) Source code/documentation consistency c3*:* All participants agree that consistency is important because documentation builds "trust" between the developer and the library. P3 elaborates *"Let's say [I start with] 100% trust the documentation is up to date and correct, and if I ever find out that it is not the case, even for one method, then I tend to kind of ignore it and start digging into the code much more."* This supports our current metric of rating the consistency between the documentation and source code.

5) Navigation score c3*:* Our intent with "Navigation score" is to capture how easy it is to navigate the documentation to find wanted information. Participants expressed interest in this metric but did not have feedback about presenting or measuring it differently. One comment all participants mentioned is that they like to see a "quick-start" guide as part of a library's documentation as quick start sections provide basic information without too much navigation into the documentation. When implementing navigation score in Section V-D, we create metric M9 to capture this.

6) Library tasks c6*:* For this component, the participants responded that knowing the tasks a library supports is important. However, they commented that there are too many tasks in the list and that it is difficult to navigate. Aside from presentation concerns, P3 states that *"Code examples...I think is the thing that I love to see the most"*, and this is because *"...descriptions I tend to get kind of confused...code examples if you run with these parameters, you get that."* In other words, code examples provide a clearer understanding of the intentions of a library, and how it accomplishes a particular task compared to textual descriptions. As such, code examples go hand in hand with knowing what tasks a library supports. This feedback supports our decision to additionally provide information about whether a task has an accompanied code example in the documented task list.

V. Implementation of Documentation Quality Summary

We now proceed to implement the metrics and their visualizations. Figure 2 shows our finalized documentation quality summary, which incorporates the interview feedback. In this section, we describe our implementation of each metric and the ground truth we used for verification.

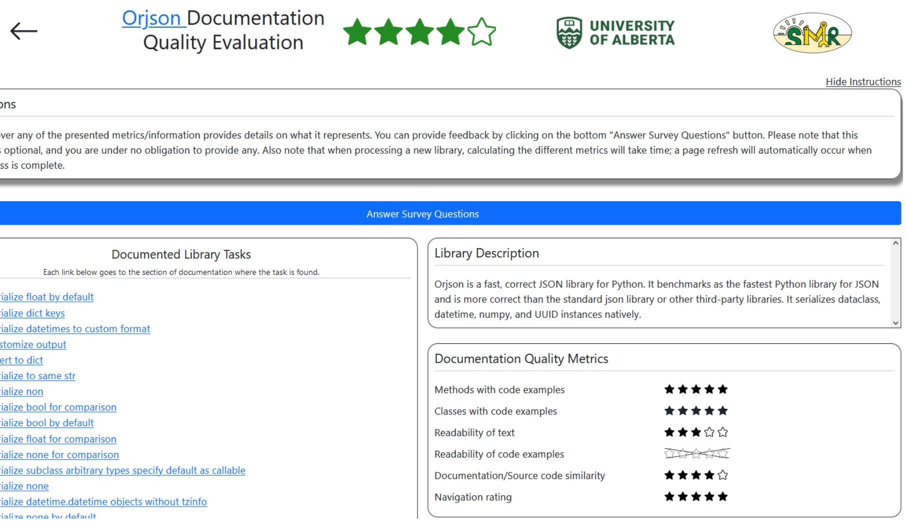
A. Documented Task List

We now discuss how we implement metric M3 which was displayed as component c6 in Figure 1. In our final implementation in Figure 2, we instead display it on the right side, as suggested, and rename it to "Documented Library Tasks". We also sort the tasks alphabetically and use the green visual checkmark to indicate the presence of code examples. To implement this component, we first extract tasks from the documentation (*task extraction* phase) and then find and link code examples to those tasks (*task linking* phase).

1) Task Extraction: To extract tasks from the documentation, we use TaskNav (short for TaskNavigator), a task extraction tool created by Treude et al. [42]. To identify tasks, TaskNav relies on analyzing the grammatical structure of a sentence to see if there is a potential task phrase. Specifically, the authors define a *task phrase* as a *"...[a] verb involved in a dependency with an object or with a prepositional phrase (or both)..."* [42]. They focus on a specific set of predefined programming verbs based on extensive testing and investigation of documentation pages. We provide the full list of considered programming verbs on our artifact page [18].

To demonstrate how TaskNav works, let us take paragraph "p-1" from Listing 1 as an example. TaskNav splits this paragraph into three sentences and extracts the following four tasks from them: (1) run series before entity building, (2) run series of TokensRegex rules, (3) specify set of TokensRegex rules, and (4) call TokensRegexAnnotator sub-annotator. There are three tasks extracted from the first sentence, because the words "run", and "specify" are in the list of considered verbs, and there is a dependency from these verbs to the objects "entity building", and "TokensRegex rules". The second sentence has the verb "call" and the object "TokensRegexAnnotator sub-annotator", leading to the fourth extracted task, while the third sentence does not have any extracted tasks.

To verify the correctness of our task extraction process, we evaluate it on eight documentation pages from four different domains and three different programming languages. We use the orjson [43] and JSON-java [44] library README pages, the Stanford NLP "Named Entity Recognition" [45] and "'Command Line Usage" [46] pages, the NLTK "parse" [47] and "tag" [48] pages, jQuery "get" [49] page, and React "Component and Props" [39] page. These eight pages contain 1,681 paragraphs (based on the HTML <p> tag). Two authors manually analyze these paragraphs and extract tasks, based on the task definition of *any usage of the library, including its API, installation instructions, and interaction with other software.* After discussing and resolving disagreements, we have a ground truth of 354 tasks. Since this is not a closed coding task, we cannot use Cohen's Kappa [23] to measure agreements. Instead, we provide the percentage of paragraphs where both authors extracted the same library tasks, e.g., if author 1 extracted tasks A, B from a paragraph, and author 2 extracted tasks A, B, C, then this paragraph would be marked as a disagreement. Our percentage agreement over the eight pages ranged from 50% to 83%, with a median of 71%.


Orjson Documentation Quality Evaluation ★★★★☆

UNIVERSITY OF ALBERTA

Hide Instructions

Instructions

Hovering over any of the presented metrics/information provides details on what it represents. You can provide feedback by clicking on the bottom "Answer Survey Questions" button. Please note that this feedback is optional, and you are under no obligation to provide any. Also note that when processing a new library, calculating the different metrics will take time; a page refresh will automatically occur when each process is complete.

Answer Survey Questions

Documented Library Tasks

Each link below goes to the section of documentation where the task is found.

- ✓ Serialize float by default
- ✓ Serialize dict keys
- ✓ Serialize datetimes to custom format
- ✓ Customize output
- ✓ Insert to dict
- ✓ Serialize to same str
- ✓ Serialize non
- ✓ Serialize bool for comparison
- ✓ Serialize bool by default
- ✓ Serialize float for comparison
- ✓ Serialize none for comparison
- ✓ Serialize subclass arbitrary types specify default as callable
- ✓ Serialize none
- ✓ Serialize datetime.datetime objects without tzinfo
- ✓ Serialize none by default

Library Description

Orjson is a fast, correct JSON library for Python. It benchmarks as the fastest Python library for JSON and is more correct than the standard json library or other third-party libraries. It serializes dataclass, datetime, numpy, and UUID instances natively.

Documentation Quality Metrics

Methods with code examples	★★★★★
Classes with code examples	★★★★★
Readability of text	★★★☆☆
Readability of code examples	~~☆☆☆☆☆~~
Documentation/Source code similarity	★★★★☆
Navigation rating	★★★★★

Fig. 2: A screenshot of the final prototype of our documentation summary page

Listing 1: Example documentation html content from https://stanfordnlp.github.io/CoreNLP/ner.html

```
<p> {% p-0 %}
  By default no additional rules are run, so leaving
     ↪ ner.additional.regexner.mapping blank will cause this phase to
     ↪ not be run at all.
</p>
<h3 ...> {% h-0 %} Additional TokensRegex Rules </h3>
<p> {% p-1 %}
  If you want to run a series of TokensRegex rules before entity building,
     ↪ you can also specify a set of TokensRegex rules. A
     ↪ TokensRegexAnnotator sub−annotator will be called...
</p>
<p> {% p-2 %} Example command: </p>
<div ...>
    <pre ...> {% pre-0 %}
      java edu.stanford.nlp.pipeline.Stanford...
    </pre>
</div>
<p> {% p-3 %} You can learn more ... </p>
```

We evaluate recall and precision of our TaskNav-based task extraction on this ground truth. We extracted 592 tasks across the eight pages, with a precision of 4% to 20% for each page, and a median precision of 14%. The recall ranges from 5% to 79%, with a median of 49%. We find that the main reason for low recall and precision is that some verbs relevant to a given domain are not included in Treude et al. [42]'s list while others that appear in a task phrase are too generic. To remedy this, we create domain-specific verb lists as well as an exclusion list of verbs that raise too many false positives. Using these modified verb lists raised our median precision to 41% and median recall to 65%.

2) Code Example Linking: After extracting tasks from paragraphs, we want to link the code examples found on the same documentation page to the paragraphs that are related to them. This is not part of Treude et al. [42]'s TaskNav tool so we design the following heuristic to link code examples to tasks. Given a code example, our linking considers only paragraphs that are (1) above the code example, (2) do not have a header in between the paragraph and code example, and (3) contain a task phrase. Our intuition is that the description of a code example (in the form of a task) usually precedes the example and is contained within the same section of the document. Thus, for each code example denoted by the `<pre` ↪ `>` tag, we traverse the HTML DOM tree upwards to find the nearest paragraph that contains a task phrase from the previous step.

In the example from Listing 1, we find the code block pre-0 and attempt to link this code example to a task in a paragraph. According to our heuristics, we ignore paragraphs p-0 and p-3. Since p-0 has header h-0 between it and the code example, and paragraph p-3 is below the example, this leaves paragraphs p-1 and p-2 as candidate paragraphs. Since paragraph p-1 has four extracted tasks while paragraph p-2 does not have any extracted tasks (see Section V-A1), we link code example pre-0 to paragraph p-1. If both paragraphs p-1 and p-2 had tasks, then we would have selected the closest paragraph to the example, i.e., paragraph p-2. Note that a code example will be linked to all tasks from the linked paragraph.

To verify the task linking technique, we again construct a ground truth using the same eight pages above. Two authors select which paragraph (if any) to link with each code example available in these documentation pages. This resulted in 227 linked code examples and paragraphs. Our agreement ranged from 64%-100% per page. We find that our linking precision ranges from 0%-100% per document, with a median of 52% while recall ranges from 0%-92% with a median recall of 42%.

One reason for false positives is that our automated technique links the example to the closest paragraph **that contains tasks and is located above it**. If the descriptive paragraph is underneath the code example or the correct descriptive paragraph did not have extracted tasks, then our technique will instead incorrectly link the example to the closest paragraph with tasks. This means that evaluating the task linking on the ground truth double penalizes the program, once for its performance on the task extraction, and again for being unable to correctly link code examples. False negatives occur

Listing 2: Code example, StanfordNLP Simple API doc

```
Sentence sent = new Sentence("your text should go here");
sent.algorithms().headOfSpan(new Span(0, 2)); // Should return 1
```

when our technique fails to link an example to a paragraph because this paragraph is above the header or below the example or does not have an extracted list. To isolate the task linking effectiveness, we evaluate again using a subset of 107 code examples whose linked paragraphs contain automatically extracted tasks. Using this subset, the task linking precision ranged from 43%-82% with a median precision of 44%, while the recall ranged from 12%-75% with a median recall of 36%.

B. Documentation and Source Code Comparison

There are four metrics from Table II that require comparing the documented APIs with the APIs offered in the actual source code: M1, M2, M6, and M7. Since some of the techniques we use to calculate these metrics are common, we discuss them together in this section.

1) Methods/Classes with Code Examples: To calculate metrics M6 and M7, displayed as "Methods with examples" and "Classes with examples", we need to find which library API methods appear in documentation code examples. We begin by extracting all code examples from the documentation page designated by the `<pre>` tag. We then filter code examples without a method call by checking for at least one open parenthesis ("`(`"). This simple heuristic allows us to support incomplete code examples and several programming languages. Listing 2 shows a code example from the Stanford NLP [50] documentation. We extract the following method calls: (1) `Sentence(``your text should go here'')`, (2) `sent.algorithms()`, (3) `headOfSpan(new Span(0, 2))`, and (4) `Span(0, 2)`.

We next extract public methods from the library's source code on GitHub by traversing the Abstract Syntax Trees (AST) of each source file in the `src/` directory or in a directory with the same name as the library. We create a database of the fully qualified name of each public method we find. We then compare the method calls extracted from the code examples in the documentation to the database of public methods. We compare method names and number of arguments/parameters to find a match. If we have multiple matches (i.e., all same name and parameters), we err on the side of precision and do not report any match. Once we find a match, we consider this library API as having a code example in the documentation. For the class level, we consider that a class has code examples in the documentation if there is at least one method from this class (including constructors) that is used in a code example in the documentation according to the above matching steps.

When displaying this metric in the summary prototype, we create the star ratings by normalizing the ratios of methods/classes with code examples to percentages out of 100, where each star in the star rating then represents 20% of methods/classes having a code example in the documentation.

To verify the correctness of our method/class linking process, we create a ground truth using the same libraries we used in the task linking verification with the exclusion of React [51], and jQuery [52], and the addition of Requests [53], QUnit [54], and jBinary [55]. We replace these libraries as the React and jQuery GitHub repositories do not store their source code in a "src" directory, or a directory sharing the library name, which are the heuristics we use to automatically identify the source code directory of a given library. We manually collect the list of library method calls used in each documentation example. To report overall numbers for a given library, we calculate the median recall and precision over all documentation snippets in that library. Our precision ranges from 10%-62% , with a median precision of 32%. The recall ranges from 21%-100% with a median of 61%.

2) Documentation/Source code similarity: Metrics M1 and M2 calculate how many methods/classes in the documentation are in the library source code, indicating consistency. This differs from the previous two metrics as we do not look for code examples; instead, we match the signatures of the classes/methods in the documentation **text** with the signatures of the classes/methods in the source code.

We first extract any inline code mentions from the documentation by parsing the HTML for `<code>` and `<dt>` HTML tags. The `<dt>` HTML tag stands for "description term" and was the other common HTML tag aside from the `<code>` tag that we found to designate inline code. Once we extract candidate code mentions from the documentation, we check for method calls in this code using the same one open parentheses heuristic above. We then use regular expressions to identify the method call and its arguments, such that we can compare it to the created source code dictionary. At the end, we calculate the metric as the percentage of method calls mentioned in the documentation that are also found in the library's source code. We normalize the resulting percentage to a star rating, where each star represents 20% consistency between the documentation and source code.

We again manually create a ground truth based on the same seven libraries used in Section V-B1. To report overall numbers for a given library, we calculate the median recall and precision over all snippets in that library. Across the same 7 libraries we evaluate on, the precision of our technique ranges from 6%-8%, with a median precision of 22%. The recall ranges from 6%-24%, with a median recall of 20%.

One issue resulting in false positives is the initial heuristic of using parentheses to find method calls. For example, the documentation text may use parentheses as a way to convey information, e.g., *": - Response.__nonzero__ (false if bad HTTP Status)"*. Our technique would incorrectly extract this method and find one parameter and match it to the method found in the source code with the same name and number of parameters.

C. Readability of text and code

We now discuss how we calculate metrics M4 and M5 related to readability, shown as "Readability of text" and

"Readability of code" in Figure 2.

1) Readability of text: Similar to existing work [12], [14], [56], [57], we use the Flesch reading ease metric [58] to calculate the readability of documentation text. For each documentation page of a library, we extract all paragraphs (designated by `<p>`) and combine them into a single text that we apply the Flesch formula [58] on. For the library as a whole, we take the average reading score of each page of the documentation, resulting in a value from 0-100, which we again normalize into a star rating.

2) Readability of code: We calculate code readability using Scalabrino et al. [35]'s code readability classifier for Java code. As part of their work, they investigated various code features (e.g., line length, number of characters), including those by Buse and Weimer [59] and designed a classifier with high accuracy. This is the only metric we have that is language specific (to Java) as we rely on their implementation. We convert the code readability score to a star rating by also treating the returned value as a percentage, as Scalabrino et al. [35]'s model returns a value from 0-100, where each star represents 20% of the code examples are readable.

D. Navigability

To objectively evaluate the navigability of software documentation, we leverage Human-Computer Interaction (HCI) research. Since most official documentation pages are online webpages, utilizing HCI research backed guidelines provides us a research-supported check list of features to evaluate library documentation navigability.

We focus on six guidelines from the Web Content Accessibility Guidelines (WCAG) 2 [60] related to best design practices for navigating a web page: (G63) Provide a site map, (G64) Provide a Table of Contents, (G125) Provide links to navigate to related Web pages, (G126) Provide a list of links to all other Web pages, (G161) Provide a search function to help users find content, (G185) Link to all of the pages on the site from the home page. We also add our own seventh check for a "quick start" guide based on feedback from our interview participants (M9). For each of these items, we use a combination of parsing and heuristics to determine whether a documentation page follows these guidelines. For example, to determine if developers implemented built-in search functionality (G161), we search the HTML for input elements with an attribute of the name *"placeholder"*, or form elements that also contain a *"search"* class. Due to space constraints, we provide the remaining heuristics on our artifact page [18]. We give a library a five star rating when its documentation has more than two navigable features, having exactly two features equates to a three star rating, one feature equates to a one star rating, and no features results in a zero star rating.

VI. Survey Evaluation of our Documentation Quality Summary

To evaluate our documentation quality summary, we run a survey with software developers, with the goal of answering the following research questions:

- RQ1. How useful is our documentation summary for assessing the documentation quality of a library?
- RQ2. Does our summary match users' library experience?
- RQ3. Where do users want to see this summary information?

A. Survey Setup & Recruitment

a) Survey Setup: We create a website that displays the documentation quality summary shown in Figure 2, while also including a survey that developers can fill to provide feedback. The landing page of our web application allows the user to choose one of the existing libraries we have information about or to analyze a new library. We populate the initial list of available libraries with the nine libraries we previously used in the different metric verifications. If the user chooses to analyze a new library, then they must provide: (1) the library name, (2) the library's main programming language, (3) the official documentation URL for the library, and (4) the GitHub repository URL (optional). Note that if no GitHub repository URL is given, the metrics requiring code analysis will not run. After selecting either of those two options, we then ask for the user's optional demographic information in terms of the years of software development experience they have and whether they are already familiar with the library they chose. The user then sees the documentation quality summary of the library they chose and can interact with it by hovering over any of the metrics or clicking on a task link, etc. If the user decides to participate in the survey, they can access the survey by clicking on the blue "Answer Survey Questions" button, which will display the following survey questions in the right margin of the page.

To answer RQ1, our survey questions ask about the usefulness of each individual metric and then the overall usefulness of the summary. For each question, we use a Likert scale [61] rating, e.g., one question we ask on the CoreNLP [50] summary page is: "How useful is having a *list of documented tasks* in *CoreNLP's* documentation?", with responses being a Likert scale from "Not useful" to "Very useful".

If the user expressed earlier that they are familiar with the library, we ask the following additional survey question to answer RQ2: "To what extent do you agree or disagree with this statement: *The documentation quality metrics represented in this summary are consistent with my experience working with* `<library name>` *(e.g., the metrics indicate low documentation quality and your experience is that this library is poorly documented, or vice versa).*", which also has Likert scale [61] responses from "Strongly disagree" to "Strongly agree".

Finally, to answer RQ3, we ask participants where they would like to see our documentation quality summary integrated. We provide a drop down list of four options: "None", "README file badge", "Package manager", and "Other", where "Other" allowed the user to input their own suggestion.

b) Participant recruitment: We recruited participants for the survey following the Snowball sampling [62] and Convenience sampling [63] methods using social media apps, such as Facebook and Twitter, and personal connections.

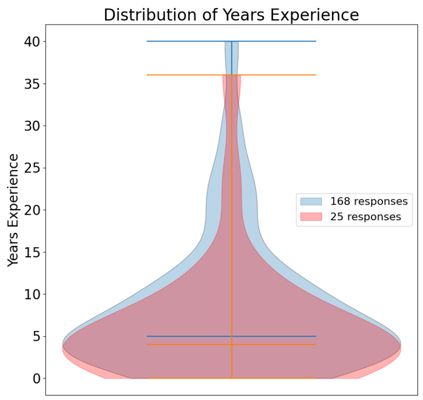


Fig. 3: Distribution of years of experience of the tool accessors and survey participants

c) Responses: Despite having 173 participants access the service, we received 26 survey responses (i.e., the participant answered at least one of the survey questions). Of these 26 responses, 25 rated every metric, while 1 response only rated the general usefulness of the tool. Figure 3 shows the distribution across years of experience for both all service accesses and survey participants. The 26 responses were made on 12 different libraries as follows. We received 15 responses on Python libraries `numpy` (10), `nltk` (1), `Tensorflow` (1), `pyppeteer` (1), `scikit-learn` (1), and `Flask` (1). We received 8 responses on JavaScript libraries `React` (4), `jQuery` (3), and `axios` (1). We also received 1 response for TypeScript (where the user treated it as a library). Finally, we received 2 responses on Java libraries `Mockito` (1) and `Soot` (1). We base our survey findings on the ratings from these 26 responses.

B. RQ1: Usefulness of our documentation quality summary

To answer RQ1, we analyze the distribution of the 25 responses on each metric, shown in Figure 4. From the median score of each metric, we see that most participants found most of the provided metrics to be useful. We can see that the 'Text Readability" metric in particular seem to be the most useful of our metrics as it has no rating of one, meaning that no participant found it to be "Not useful". When viewing the other swells of the violin plot, most metrics have the majority of the responses around a rating of four or five, with the exception of the "Documentation/Source Code similarity" metric, which has an even distribution across the different rating values. Overall, despite some differences in opinion about the individual metrics, the overall rating for the usefulness of our documentation quality summary (General rating shown at the top of Figure 4) has a median of 4, with most responses around that median.

C. RQ2: Matching Library Usage Experience

There are 21 responses indicating that the participant is familiar with the library they are responding about, which we use to answer RQ2. Figure 5 shows these participants' ratings on whether the summary matches their experience using this

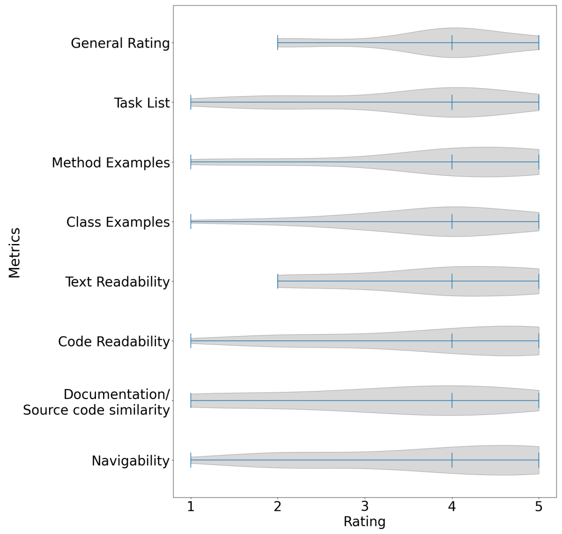


Fig. 4: The distribution of ratings for each metric in our documentation quality summary

library. We find that the responses are a bit controversial with some participants agreeing with our documentation quality assessment while others not. The free-form comments give us some insights into this controversy where participants found that some metrics reflected their experience while other metrics did not, resulting in them providing a low rating for this question. For example, one participant responded that *"My experience with React's documentation has been overall pleasant. While I agree with the navigation rating, I disagree with the documentation quality metric for "Readability of text" and "Readability of code examples"..."* Out of the 12 evaluated libraries, 10 participants rated 7 of the library documentation quality summaries as matching their experience using that documentation.

D. RQ3: Integration of documentation quality summary

Finally, to answer RQ3, we ask participants where they would like to see library documentation quality summaries. We find that 17 (65%) participants want to see this information as a badge in a library's ReadMe file, 6 (23%) participants want to see it as part of the library's information in the package manager, and one (4%) participant wants to see it in both places. Only two of the participants say they would not like to see this information any where. Integration of this documentation quality summary into alternative formats allows developers to see high level information about the documentation quality at a glance.

VII. Discussion & Implications

Our survey results suggest that developers appreciate being able to evaluate a library's documentation page. Our work provides this service, based on systematic steps of studying the literature and implementing and verifying metrics, including intermediate feedback from software developers.

Our documentation quality summary can be integrated with other tools evaluating or comparing libraries. For example, our

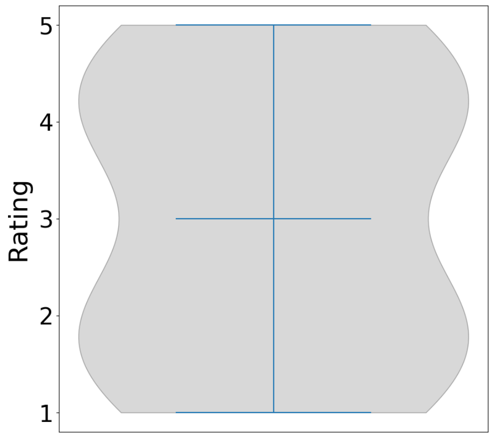


Fig. 5: The distribution of ratings for 21 participants familiar with the library they were viewing, of whether the documentation quality summary matches their expectations

work could be integrated with the previous work done by De la Mora and Nadi [2]. Their goal was to help developers compare libraries across different characteristics to help with selecting the most suited one for their needs. Adding documentation quality to their library comparison website will help developers understand if there will be enough information for them to use to learn the library. In fact, in their survey of 61 developers, De la Mora and Nadi [2] found documentation quality to be the most demanded metric from their participants.

Based on RQ3, our documentation summary could also be integrated into other formats that provide quick evaluation of a library, such as README badges or even as part of package managers (e.g., npms.io).

VIII. Threats to Validity

a) Construct Validity: Each documentation aspect can be measured in multiple ways. We used the GQM [17] method to derive metrics that can be objectively automated, but we do not claim that these are the only metrics that can be used to measure a given aspect. Thus, our evaluation evaluates the usefulness of the metrics we derived, rather than the usefulness of a particular aspect, which has already been determined in the literature (Section II). We rely on manual validation to create the ground truth for the different metrics. To overcome any subjectivity or error, we define the goal of each manual analysis and have two authors construct the ground truth.

b) Internal Validity: We find that for the libraries participants evaluated, our implementation often displays the "Documentation/Source code similarity" metric as zero out of five stars. This is because we do not display partial stars, meaning that a metric needs to meet the "20%" threshold for at least one star to appear. We believe that the empty stars led to the controversy/poor rating of this metric in RQ1.

Another threat is the low precision/recall of some of our metrics. This could be mitigated in the future by, for example, using language-specific parsers instead of regular expressions for the metrics related to retrieving and comparing method signatures. However, the purpose of our survey is not only to gauge the accuracy of the metrics in our summary tool, but also the developer response of the usefulness and desire of having such a tool to rate documentation quality. We thus chose to conduct the survey to get this important feedback; we plan to continue to improve our metrics.

c) External Validity: To improve task extraction, we create a verb list per domain. Although this list is not based on an exhaustive analysis of all libraries in a given domain, it is based on 148 different documentation pages from eight libraries and has increased the precision of the task extraction by 27% on average. We share our verb list on our artifact [18] page for further validation or extension. Automatically creating comprehensive verb lists for each domain is a challenging task, opening up opportunities for additional research.

Our results are based on responses from only 26 developers, despite having 173 people access our survey link and receiving positive endorsements on social media. We believe that the small number of recorded responses is due to a technical bug we had, which resulted in survey responses not being stored. Unfortunately, this bug did not manifest itself in our pilot study with 5 participants. That said, the high number of developers who accessed our service does suggest that developers are generally interested in tooling that allows them to assess the documentation quality of a library. Overall, the 26 responses we use cover a wide range of experienced developers over 12 different libraries in each of the 3 supported languages. Figure 3 compares the distribution of years of experience for survey accessors and respondents; a Kolmogorov–Smirnov test [64] shows no statistically significant differences.

IX. Conclusion

This paper presented metrics and an early prototype implementation to measure documentation quality. We create this tooling as a service that provides a visual summary consisting of values for 9 metrics that reflect the quality of a given documentation page and its subpages. We identified these 9 metrics through systematically studying the literature to identify documentation aspects and then used the GQM [17] paradigm to derive metrics. We conducted an initial validation of these metrics and their visualization through an interview study with three developers. We then implemented and verified all our metrics before conducting a survey with 26 participants to evaluate the usefulness of our metric and documentation quality summary. Our results show that participants find our metrics useful and 65% would like to see this summary incorporated as a badge in a library's ReadMe page. As future work, we hope to create a badge for our summary as well as to integrate it into existing library comparison tooling [2]. All of our data and code are available online on our artifact page [18].

References

[1] J. Rowley, "Guidelines on the evaluation and selection of library software packages," in *Aslib proceedings*. MCB UP Ltd, 1990.

[2] F. L. de la Mora and S. Nadi, "An empirical study of metric-based comparisons of software libraries," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 22–31. [Online]. Available: https://doi.org/10.1145/3273934.3273937

[3] F. Thung, "Api recommendation system for software development," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 896–899.

[4] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[5] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," ser. DocEng '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 26–33. [Online]. Available: https://doi.org/10.1145/585058.585065

[6] G. Garousi, V. Garousi-Yusifoğlu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and usefulness of technical software documentation: An industrial case study," *Information and Software Technology*, vol. 57, pp. 664–682, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491400192X

[7] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[8] G. Garousi, V. Garousi, M. Moussavi, G. Ruhe, and B. Smith, "Evaluating usage and quality of technical software documentation: An empirical study," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 24–35. [Online]. Available: https://doi.org/10.1145/2460999.2461003

[9] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, benefits and quality of software development documentation: A systematic mapping," *Journal of Systems and Software*, vol. 99, pp. 175–198, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002131

[10] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, ser. SIGDOC '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 68–75. [Online]. Available: https://doi.org/10.1145/1085313.1085331

[11] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 245–256. [Online]. Available: https://doi.org/10.1145/3368089.3409711

[12] A. Forward, *Software documentation: Building and maintaining artefacts of communication.* University of Ottawa (Canada), 2002.

[13] A. Wingkvist, M. Ericsson, R. Lincke, and W. Löwe, "A metrics-based approach to technical documentation quality," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, 2010, pp. 476–481.

[14] L. Aversano, D. Guardabascio, and M. Tortorella, "Evaluating the quality of the documentation of open source software." in *ENASE*, 2017, pp. 308–313.

[15] H. Zhong and Z. Su, "Detecting api documentation errors," *SIGPLAN Not.*, vol. 48, no. 10, p. 803–816, 2013. [Online]. Available: https://doi.org/10.1145/2544173.2509523

[16] S. Lee, R. Wu, S.-C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 653–675, 2021.

[17] R. van Solingen (Revision), V. Basili (Original article, 1994 ed.), G. Caldiera (Original article, 1994 ed.), and H. D. Rombach (Original article, 1994 ed.), *Goal Question Metric (GQM) Approach*. John Wiley & Sons, Ltd, 2002. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof142

[18] "Online artifact page," https://figshare.com/articles/software/Evaluating_Software_Documentation_Quality/22177961.

[19] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: The practitioners' perspective," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 590–601.

[20] A. S. M. Venigalla and S. Chimalakonda, "Understanding emotions of developer community towards software documentation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, 2021, pp. 87–91.

[21] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[22] A. Jazzar and W. Scacchi, "Understanding the requirements for information system documentation: An empirical investigation," in *Proceedings of Conference on Organizational Computing Systems*, ser. COCS '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 268–279. [Online]. Available: https://doi.org/10.1145/224019.224048

[23] C. Treude, J. Middleton, and T. Atapattu, "Beyond accuracy: Assessing software documentation quality," *CoRR*, vol. abs/2007.10744, 2020. [Online]. Available: https://arxiv.org/abs/2007.10744

[24] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, "Api documentation and software community values: A survey of open-source api documentation," in *Proceedings of the 31st ACM International Conference on Design of Communication*, ser. SIGDOC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 165–174. [Online]. Available: https://doi.org/10.1145/2507065.2507076

[25] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[26] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software*, vol. 82, no. 6, pp. 981–992, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121208002793

[27] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1199–1210.

[28] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 27–37.

[29] F. Bachmann, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers, R. Nord, and R. Little, "Software architecture documentation in practice: Documenting architectural layers," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2000.

[30] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, "A decade of code comment quality assessment: A systematic literature review," *Journal of Systems and Software*, p. 111515, 2022.

[31] J. M. Kleinberg *et al.*, "Authoritative sources in a hyperlinked environment." in *SODA*, vol. 98. Citeseer, 1998, pp. 668–677.

[32] R. Flesch, "Flesch-kincaid readability test," *Retrieved October*, vol. 26, no. 3, p. 2007, 2007.

[33] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[34] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[35] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[36] T. Comber, "Building usable web pages: An hci perspective," in *Proceedings of the First Australian World Wide Web Conference*. Norsearch, Ballina, Australia, 1995, pp. 119–124.

[37] C. Nicolle and J. Abascal, *Inclusive design guidelines for HCI*. CRC Press, 2001.

[38] W3C, "How to meet wcag (quick reference)." [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/?showtechniques=245#multiple-ways

[39] "Components and props." [Online]. Available: https://reactjs.org/docs/components-and-props.html

[40] Y. Zhou, X. Yan, T. Chen, S. Panichella, and H. Gall, "Drone: A tool to detect and repair directive defects in java apis documentation," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 115–118.

[41] R. Longhurst, "Semi-structured interviews and focus groups," *Key methods in geography*, vol. 3, no. 2, pp. 143–156, 2003.

[42] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 565–581, 2015.
[43] Ijl, "Ijl/orjson: Fast, correct python json library supporting dataclasses, datetimes, and numpy." [Online]. Available: https://github.com/ijl/orjson
[44] Stleary, "Json in java." [Online]. Available: https://github.com/stleary/JSON-java
[45] "Named entity recognition." [Online]. Available: https://stanfordnlp.github.io/CoreNLP/ner.html
[46] "Command line usage." [Online]. Available: https://stanfordnlp.github.io/CoreNLP/cmdline.html
[47] "nltk.parse package." [Online]. Available: https://www.nltk.org/api/nltk.parse.html
[48] "nltk.tag package." [Online]. Available: https://www.nltk.org/api/nltk.tag.html
[49] "jquery.get." [Online]. Available: https://api.jquery.com/jQuery.get/
[50] "Overview." [Online]. Available: https://stanfordnlp.github.io/CoreNLP/
[51] "React." [Online]. Available: https://reactjs.org/
[52] "jquery." [Online]. Available: https://jquery.com/
[53] "Requests." [Online]. Available: https://requests.readthedocs.io/en/latest/
[54] "Qunit." [Online]. Available: https://qunitjs.com/
[55] "jbinary." [Online]. Available: https://github.com/jDataView/jBinary
[56] G. Hargis, "Readability and computer documentation," *ACM J. Comput. Doc.*, vol. 24, no. 3, p. 122–131, 2000. [Online]. Available: https://doi.org/10.1145/344599.344634
[57] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IWPSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 4–10. [Online]. Available: https://doi.org/10.1145/1294948.1294952
[58] R. Flesch, "A new readability yardstick." *Journal of applied psychology*, vol. 32, no. 3, p. 221, 1948.
[59] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.
[60] "How to meet wcag (quick reference)," 2008. [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/?showtechniques=245#multiple-ways
[61] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.
[62] L. A. Goodman, "Snowball sampling," *The annals of mathematical statistics*, pp. 148–170, 1961.
[63] P. Sedgwick, "Convenience sampling," *Bmj*, vol. 347, 2013.
[64] D. A. Darling, "The kolmogorov-smirnov, cramer-von mises tests," *The Annals of Mathematical Statistics*, vol. 28, no. 4, pp. 823–838, 1957. [Online]. Available: http://www.jstor.org/stable/2237048