What the Fix? A Study of ASATs Rule Documentation
Corentin Latappy, Thomas Degueule, Jean-Rémy Falleri, Romain Robbes,
Xavier Blanc, Cédric Teyton

To cite this version:

HAL Id: hal-04450208
https://hal.science/hal-04450208
Submitted on 12 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
What the Fix? A Study of ASATs Rule Documentation

Corentin Latappy, Thomas Degueule, Jean-Rémy Falleri, Romain Robbes, Xavier Blanc, and Cédric Teyton

ABSTRACT
Automatic Static Analysis Tools (ASATs) are widely used by software developers to diffuse and enforce coding practices. Yet, we know little about the documentation of ASATs, despite it being critical to learn about the coding practices in the first place. We shed light on this through several contributions. First, we analyze the documentation of more than 100 rules of 16 ASATs for multiple programming languages, and distill a taxonomy of the purposes of the documentation—What triggers a rule; Why it is important; and how to Fix an issue—and its types of contents. Then, we conduct a survey to assess the effectiveness of the documentation in terms of its goals and types of content. We highlight opportunities for improvement in ASAT documentation. In particular, we find that the Why purpose is missing in half of the rules we survey; moreover, when the Why is present, it is more likely to have quality issues than the What and the Fix.

CCS CONCEPTS
- Software and its engineering → Software libraries and repositories; Software maintenance tools; Software design engineering.

KEYWORDS
software quality, automatic static analysis tools, linters, documentation

We argue that this documentation is essential for improving developers’ knowledge and skills, as well as promoting compliance with the rules. Even though warning notifications have already been studied [7, 16], empirical evidence shows that their quality is not ideal [7, 17, 18]. Furthermore, to the best of our knowledge, there is no comprehensive study on the reference documentation of ASATs rules, which developers look for after the initial warning. We believe that the reference documentation of ASATs rules is not a classical piece of software documentation, such as an API documentation [19]. Indeed, ASATs rule documentation is challenging as it needs to explain issues regardless of their particular context in such a manner that developers are able later to understand them in their own context. Another difficult point is that ASATs rules are sometimes subjective, for instance those related to the use of goto statements [20] or the classical dilemma of choosing between snake and camel case [21]. Thus, little is known about how to best document ASAT rules. This lack of knowledge is damaging when ASATs such as Semgrep allow developers around the world to effortlessly create and share hundreds of rules.

In this article, we explore the current state of the ASATs’ rules documentation and contrast it with developers’ expectations, in issues with the source code, such as not following best practices, using error-prone constructs, and detecting security or performance problems [1, 4–7]. When such an issue is detected, a warning is issued to the developers of the project so that they can inspect the incriminated code and fix the issue if necessary. Issues are generally defined by rules, a popular example being the eqeqeq rule in ESLint JavaScript’s ASAT that advises developers to use === instead of == to avoid type coercion errors.

Even though these tools are prone to issue false-positive warnings [8–10] and their effect on software quality is still a matter of debate [11–13], one positive aspect raised by the use of ASATs is their ability to improve the developer’s knowledge and skills and to support the onboarding of newcomers [14, 15]. In this article, we focus on the following scenario of ASATs: a developer encounters a warning for the first time and wants to learn more about it. Usually, warnings raised by ASATs within the IDE or within the terminal as a result of the use of its command line interface only contain limited information [7, 16]. To further explain a given rule, ASATs generally offer more extensive documentation via formatted documents available online.

1 INTRODUCTION
Automatic Static Analysis Tools (ASATs, sometimes called linters) [1] are very popular quality insurance tools used in a quarter [2] to half of software projects [3]. According to the https://analysis-tools.dev website, there are more than 600 ASATs. The core principle behind these tools is to scan a code base looking for evidence of potential
order to extract best practices with the goal to increase the quality of this documentation. We start with an empirical study of how real-world ASAT rules are documented. Section 2 details our study of more than 100 rules across 16 ASATs, covering 7 programming languages (plus two polyglot ASATs). Through an iterative analysis of these rules, we derive a nomenclature covering 15 documentation attributes in three themes. In Section 3, we distill our nomenclature in a simple taxonomy of documentation purposes (What triggers the rule, Why it is important, and how to Fix the issue) and content types (Text, Code, and Hyperlinks). Only half of the rules we analyze have a Why purpose.

Then, we use our taxonomy to contrast the documentation of 12 real-world rules with developer expectations via a survey. Section 4 presents the survey in which 85 respondents evaluated the rules 298 times. Among other findings, developers highlight quality issues with the documentation of the Why purpose, and emphasize the pedagogical aspects and the need for conciseness in ASAT documentation. To close the paper, we discuss the limitations of this study (Section 5), other studies of ASATs and documentation (Section 6), before concluding (Section 7). A replication package including raw data, the nomenclature and taxonomy, and survey results is available online1 and on Zenodo [22].

2 A NOMENCLATURE FOR RULE DOCUMENTATION

The first step of our study is to create a nomenclature based on the documentation of the rules provided by ASATs. A nomenclature is a classification tool that provides a structured and systematic way of naming and referring to a wide range of objects. This nomenclature will help us define the types of information we find in a rule’s documentation and group them by purpose. To build it, we use a three-step process described next: (1) we select a total of 16 diverse ASATs that provide documentation for their rules; (2) we iteratively build a corpus of 119 rules, for which we code the documentation concepts we encounter to support their comparison; and (3) we compare and analyze the rule documentations to provide the expected nomenclature, which covers 15 different concepts.

2.1 Selecting ASATs

A GitHub project listing ASATs2 references over 600 different tools; choosing a reasonable subset is both necessary and challenging. A first hard requirement is that we only include ASATs focusing on code rule enforcement, leaving out for instance code beautifiers. A second hard requirement is that we only include ASATs that provide a website presenting the rules documentation. Finally, we select ASATs emphasizing both diversity and popularity. Since we want our nomenclature to be used beyond our study, regardless of ASAT, we include a diverse set of ASATs targeting different programming languages. Finally, we want to bias our corpus toward popular ASATs as we postulate that popular ASATs have more odds of including well-thought-out and comprehensive documentation.

We define two criteria to judge diversity and one criterion to judge popularity. The two diversity criteria we propose are: that the ASATs covers most of the popular programming languages3, and to have at least two ASATs per programming language covered; we also include ASAT that supports multiple languages to account for this category. The popularity criterion we propose is that ASATs must have a GitHub repository with at least 1000 stars.

Starting with the ASAT list, we select the 16 ASATs in table 1 using our criteria and the domain expertise of our industrial partner. The only exception to the popularity criterion is Gendarme, which we pick to increase diversity: it is the official ASAT for the alternative Mono C# implementation. We are aware that this selection is somewhat arbitrary and that another selection could have been made with the same criteria; we discuss this issue in section 5.

2.2 Coding Documentation Concepts

Our selection of ASATs offer rule documentation that differs in structure and in content, and this can even happen between two rules from the same ASAT. To compare ASAT documentations, we identify and align all the concepts used in the documentation.

For example, fig. 1 displays the documentation for the rule pointless-statement provided by PyLint. Using visual and graphical cues on this documentation (headers, line breaks, boxes, images, etc.), we extract and code four concepts: an emitted message (error message when the rule is violated), a description, a problematic code, and a correct code. If we now want to compare this documentation with another rule, we need to align their documentation concepts. This alignment will, e.g., reveal which concepts are present in both documentations, or which ones are present in only one rule.

Each ASAT provides many rules. We use a two-step process to first code and then reconcile the documentation concepts. More precisely, we build an independent coding for each ASAT iteratively and then reconcile them into a single and global coding for rules documentation. For each ASAT, we pick up rules at random, look at the documentation for visual and graphical cues, and add documentation concepts if they were not identified before. We repeat this until saturation—analyzing 5 successive rules without discovering a new concept. Using this process, we analyze at least 6 rules per ASAT; table 2, column 3, counts the rules we inspect per ASAT.

The reconciliation of the documentation concepts identified in each ASAT was done manually by the authors during a harmonization session. For instance, we consider that the Message emitted

\[1\]https://icpc2024-asats.github.io

\[2\]https://github.com/analysis-tools-dev/static-analysis, 12K stars

\[3\]https://madnight.github.io/githut/#/pull_requests/2021/4
What the Fix? A Study of ASATs Rule Documentation
ICPC ’24, April 15–16, 2024, Lisbon, Portugal

pointless-statement / W0104

Message emitted:

Statement seems to have no effect.

Description:

Used when a statement doesn’t have (or at least seems to) any effect.

Problematic code:

```python
[1, 2, 3] # pointless-statement
```

Correct code:

```python
NUMBERS = [1, 2, 3]
print(NUMBERS)
```

Figure 1: Documentation of the rule pointless-statement from Pylint

A concept in Pylint rule of fig. 1 is highly similar to the Message output concept identified in Cppcheck rules. We therefore reconcile these two concepts and code them with Error Output.

As a result, we build a corpus of 119 rules, with which we identify 15 documentation concepts across ASATs (see table 2). Documentation consistency vary: depending on the tool, we need between 6 and 10 rules to reach saturation.

2.3 The Nomenclature

After identifying documentation elements in the previous phase, to finalize our nomenclature, we grouped the documentation concepts we encountered into different themes to clarify their role. This classification was made by the authors following an approach similar to open card sorting [23]. We obtained the following three themes:

- **Comprehension**, which contains all the concepts identifying parts of the documentation that help to understand the rule.
- **Usage**, which contains all concepts identifying parts explaining how to correctly configure the rule for a given project.
- **Metadata**, which contains all the concepts identifying ASAT-specific information (such as organizational scheme or rules source code).

Our nomenclature is presented in table 2 with 15 documentation concepts grouped into three themes. For the sake of readability, we present the results by ASAT, even if there are differences between rules within the same ASAT. For each ASAT and concept, table 2 shows the percentage of the concept’s presence across all analyzed rules (the definition of each concept and the full per-rule version are available in our replication kit).

The first point of interest is that there is no single documentation concept that can be found in all ASATs. The two most common concepts are description (100% in all but one ASAT, Roslynator) and code example (70–100% in all but one ASAT, SpotBugs). This presence is mirrored in the rules: 109 out of 119 rules have descriptions, and 104 out of 119 have code examples. We were surprised by the lack of descriptions for Roslynator, even since every other ASAT had a description: for this ASAT, the title of the rule acts as a description.

Other concepts are sparser, with 40 rules out of 119 (6 ASATs) that include a severity, and 36 out of 119 that include further information (12 ASATs). Some concepts are very sparse, such as IDE Fix (2 rules out of 119) and compatibility (4 out of 119).

The second point of interest is that, apart from the description, the code example, and the further information concepts, few ASATs share the same concepts. We note that the average number of concepts per ASAT is 5.1, with a median of 4.5. ESLint is the exception, with 11 concepts used, albeit with only three concepts used consistently (description, since, and rule definition).

The final point of interest is that few ASATs are consistent in their use of the documentation elements. Only ESLint is completely consistent, while ESLint is the most inconsistent.

Finding #1: Projecting our nomenclature onto the rules of the 16 ASATs we selected clearly reveals the differences in terms of rule documentation. This reinforces the need to define a more abstract taxonomy and to carry out a survey of developers to better understand their expectations and calls for action, which is the purpose of the following sections.

3 TAXONOMY OF CONTENT PURPOSES AND TYPES

In this section, we dive deeper into the actual content of rules documentation to better understand their purpose. We focus on the nomenclature’s Comprehension theme as, we are interested in the developer’s comprehension of ASAT rules and the best practices they describe. In contrast, the nomenclature’s Usage theme relates to the usage of the ASAT (e.g., how to configure a rule), while Metadata mainly contains ASAT-specific information (such as categories of rules or source code).

The Comprehension theme consists of four terms: Code Example, Description, Further Information, and When Not To Use It (Table 2); we focus on this theme in the following. When looking at the data, one quickly realizes that these umbrella terms conceal a rich underlying diversity of purposes and content types. For instance, Description content sometimes highlights the rationale for a particular rule; at other times, it describes how to identify code that breaches the rule. Similarly, Code Example snippets may present compliant code, non-compliant code, or a mix thereof. The content also uses a combination of text, hyperlinks, and source code.

Section 3.1 presents the methodology we follow to extract and consolidate this information; Section 3.2 describe how we validate the resulting taxonomy internally (the survey in Section 4 provides additional validation). Finally, Section 3.3 presents the results of applying our taxonomy to the rules of Table 2.

3.1 Extraction

To better understand what is the purpose and the types of content used in ASAT rule documentation, we employ an informal open
Table 2: Percentage of presence of documentation concepts for each ASAT

<table>
<thead>
<tr>
<th>Language</th>
<th>ASAT</th>
<th># Rules</th>
<th>Comprehension</th>
<th>Usage</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>OCLint</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Cppcheck</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>83</td>
</tr>
<tr>
<td>C#</td>
<td>Gendarme</td>
<td>9</td>
<td>89</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Roslynator</td>
<td>10</td>
<td>100</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Java</td>
<td>Checkstyle</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>67</td>
</tr>
<tr>
<td>JS</td>
<td>ESLint</td>
<td>10</td>
<td>70</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>RSLint</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>38</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP CS Fixer</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>29</td>
</tr>
<tr>
<td>Java</td>
<td>Psalm</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>14</td>
</tr>
<tr>
<td>Python</td>
<td>Pylint</td>
<td>10</td>
<td>100</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Flake8</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>83</td>
</tr>
<tr>
<td>Ruby</td>
<td>RuboCop</td>
<td>9</td>
<td>89</td>
<td>100</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Brakeman</td>
<td>7</td>
<td>71</td>
<td>100</td>
<td>57</td>
</tr>
<tr>
<td>Multi</td>
<td>Sengteg</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>SonarLint</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>119</td>
<td>87</td>
<td>92</td>
<td>30</td>
</tr>
</tbody>
</table>

card-sorting [23] methodology coupled with a saturation process similar to the one used in Section 2. The objective is twofold: identify how the content is materialized in the documentation (e.g., text, source code, images, etc.), and what is the purpose of the content.

Traditional card sorting requires printing fragments of the documentation content on cards and physically regrouping them by common themes. Unfortunately, the large amount of content present in our sample (more than a hundred rules) makes this methodology impractical.

To simplify the process, we go over rules incrementally in an arbitrary order, limiting the open card sorting process to the content of these rules relevant to the Comprehension theme. We end the process when no new theme emerges after five successive rules.

Two of the authors, who participated also in the construction of the nomenclature and were already familiar with the content of the documentation, conducted the open card sorting during a collaborative session, in which they reviewed a few dozen rules. At the end of the session, they obtained the following taxonomy:

- **Purpose**
  - **What**: What triggers the activation of this rule and how to recognize violating code?
  - **Why**: Why does this rule matter, and why should it be enforced?
  - **Fix**: How should violating code be fixed to comply with this rule?

- **Content type**
  - **Text**: Free-form prose text
  - **Code**: Source code written in (one of) the programming language targeted by the ASAT, possibly including some prose embedded as comments
  - **Hyperlink**: Hyperlinks to other documentation, web pages, PDFs, etc.

Figure 2 shows an example rule from Checkstyle with its purposes and content types. Its description employs a mix of Text and Hyperlink to document the What and Why purposes. The Code, on the other hand, documents the What and Fix purposes via examples of compliant and non-compliant code.

3.2 Validation

We validate the completeness and objectivity of our taxonomy by calculating the agreement of independently annotating a set of rules. Note that this is an internal validation of our taxonomy; our survey (Section 4) validates it with external respondents.

The first author selects 12 rules from the considered ASATs, making sure to select non-trivial6 rules covering the main categories of ASAT rules identified by Vassolo et al. [24]: naming and style, correctness, performance, and security. The first author then manually removes (when present) from their documentation the content related to Usage and Metadata—irrelevant to our taxonomy—keeping only the content related to Comprehension. Finally, the first author annotates the rules by highlighting the elements that address the What, Why, and Fix purposes, as well as the content types used, as shown in Figure 2.

6An example of a trivial rule is https://rslint.org/no-await-in-loop/
Table 3: Percentage of presence of each taxonomy purposes regarding the type of content for each ASAT

<table>
<thead>
<tr>
<th>Language</th>
<th>ASAT</th>
<th># Rules</th>
<th>Text</th>
<th>Code</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>What (%)</td>
<td>Why (%)</td>
<td>Fix (%)</td>
</tr>
<tr>
<td>C / C++</td>
<td>OCtLint</td>
<td>6</td>
<td>100</td>
<td>33</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Cppcheck</td>
<td>6</td>
<td>100</td>
<td>83</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Gendarme</td>
<td>9</td>
<td>100</td>
<td>67</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Roslynator</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>Checkstyle</td>
<td>6</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>SpotBugs</td>
<td>6</td>
<td>100</td>
<td>83</td>
<td>100</td>
</tr>
<tr>
<td>JS / TS</td>
<td>ESLint</td>
<td>10</td>
<td>100</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>RSLint</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP CS Fixer</td>
<td>7</td>
<td>100</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Python</td>
<td>Pylint</td>
<td>10</td>
<td>100</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Flake8</td>
<td>6</td>
<td>100</td>
<td>17</td>
<td>83</td>
</tr>
<tr>
<td>Ruby</td>
<td>RuboCop</td>
<td>9</td>
<td>100</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>Multi</td>
<td>Brakeman</td>
<td>7</td>
<td>100</td>
<td>86</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Sengrep</td>
<td>6</td>
<td>100</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>SonarLint</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>83</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>119</td>
<td>93</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

The twelve rules are split randomly among three of the remaining authors, who follow the same rating process on four rules each, ensuring that each rule is annotated by two independent raters. We evaluate our taxonomy on these rules by assessing i) to what extent the What, Why, and Fix purposes are necessary and sufficient to rate all the documentation content (completeness), and ii) to what extent independent raters reliably agree on the rating of documentation content (objectivity).

Completeness. All raters used all purposes on all the rules. All raters used at least one purpose on all Text content, most Hyperlinks, and the majority of Code snippet. Some parts of the Code snippets were not rated as they do not relate to any purpose but rather serve (for instance the class declaration in Figure 2). One rater found that some Hyperlinks were too general to be actionable and that it was not clear how they relate to the rule documentation (e.g., a link to a list of the ten most critical vulnerabilities in web applications).7

Objectivity. We measure the agreement among raters using Cohen’s kappa for the textual content, at the word level with regard to the purposes. (the content type being completely objective). Cohen’s kappa is suitable for this situation as there are two raters per word, and each word is rated with a single purpose. For the Code and Hyperlinks, however, the same link or snippet may be associated with multiple labels. We use the better-suited weighted Fleiss’ kappa with a weight computed using the MASI distance between sets of labels [25].

- **Text**: we obtain kappa values of 0.3, 1, and 0.741 between the first author and the three other raters, indicating a rather strong agreement on textual content, except with one rater. The rater with the lowest agreement value performed its rating at the sentence level, while the other raters coded at the finer-grained level of individual words.
- **Code**: the kappa values are 1, 0.71, and 1, indicating a very strong agreement on this type of content.
- **Hyperlinks**: we obtain kappa values of 0.04, 0.33, and 0.14, suggesting a low agreement between the raters for this type of content. The disagreements are due to: i) some raters only rated using the context in which the link was used, without looking at its content, to perform the rating ii) some raters did not assign any rating to very general links, while others assigned all the possible ratings.

Since the agreement is overall high, with clear reasons for the disagreements we observe, our results indicate that our taxonomy is suitable for classifying ASAT documentation content.

3.3 Results

The first author applies the taxonomy to the 119 rules of Table 2 to highlight the content types and purposes for all documentation content pertaining to the Comprehension category. With regard to the objectivity issues identified in Section 3.2, the first author applies a fine-grained strategy to rate the Text content (rating at the word level), and an optimistic strategy to rate Hyperlinks (assigning multiple purposes to general documents using their content). Table 3 shows the percentage of rules, for each ASAT, that document the What/Why/Fix purposes for Text, Code, and Hyperlinks.

**Purposes.** Out of the 119 analyzed rules, 119 (100%) document the What purpose, 60 (50%) document the Why purpose, and 92 (77%) document the Fix purpose, regardless of the content type. Breaking down by content type, we see:

- **What**: 110 rules document it with Text (93%), 108 with Code (91%), and 69 with Hyperlinks (58%).
- **Why**: 60 rules document it with Text (50% overall, 100% when present), 6 with Code (5% overall, 10% when present), and 14 with Hyperlinks (12% overall, 23% when present).
While all ASATs document the What purpose, one ASAT does not document the Fix purpose (OCLint) and some rarely document it (Cpcheck, Psalm). Finally, some ASATs do not document the Why purpose (Roslynator, PHP CS Fixer) and some rarely document it (Psalm, Pylint, Flake8, OCLint).

Finding #2: ASAT documentation has three main purposes: while the What is systematically documented, the Fix is often documented (77%), and the Why is documented only half of the time (50%). The What is documented with Text (92%), Code (91%) or Hyperlinks (58%), the Why with Text (100%), Hyperlinks (23%) or Code (10%), and the Fix with Code (90%), Text (69%) or Hyperlinks (9%).

4 QUESTIONNAIRE SURVEY

In this section, we evaluate whether the documentation of ASAT rules meets the expectations of their users, as well as validating the taxonomy with said users. Specifically, we assess whether the purposes documented in the rules and their incarnation as text, source code, and hyperlinks satisfy the developers facing them. To answer these questions, we design an anonymous questionnaire survey with a mix of open-ended and closed-ended questions shared with industrial partners and fellow researchers. Section 4.1 presents the design of our survey and Section 4.2 gives an overview of the participants and the analysis methodology. Section 4.3 details the qualitative results obtained for closed-ended questions and Section 4.4 the qualitative results emerging from the open-ended questions. The survey, responses, and plots we discuss in this section are available on an interactive companion webpage.8

4.1 Survey Design

The survey opens with a welcome message detailing its goals, authors, estimated completion time, and data policy. The remainder of the survey revolves around four parts: developer profile, taxonomy evaluation, rules analysis, and general feedback. Table 4 shows the questions in the survey.

Developer profile. To establish the profile of our participants, we ask about their experience as developers and which programming languages they use regularly. Participants may pick between four groups: Novice (0 to 4 years of experience), Junior (5–9), Confirmed (10–19), and Senior (20+). They select or enter their preferred programming languages from an open-ended list. As our study focuses explicitly on ASATs, we ask the participants about their experience with linters.9 The first question asks whether they know what a linter is, the second whether they use linters on some of their projects, and the last one which linters they use (picked from the 16 ASATs we study in this paper or entered manually).

Taxonomy evaluation. In this part, participants are shown a screenshot of the FetchEnvVar rule from RuboCop10 which serves as an illustration to the terminology we use in the survey (linter, rule, compliant code, non-compliant code). Then, we introduce the participants to the terms of our taxonomy and their definition: purposes (What/Why/Fix) and content types (Text, Code, Hyperlinks). We then ask the participants to evaluate the importance of each purpose in the documentation of ASAT rules. We employ an asymmetric survey response scale inspired by Kano et al. [26] and adapted by Begel et al. for software engineering [27]: Essential, Worthwhile, Unimportant, Unwise, I don’t understand. For each purpose, we include an additional open-ended question asking why its presence in the documentation is or is not important. The final open-ended question asks the participants whether ASATs should document additional purposes, which we may have missed in our taxonomy.

Rules analysis. In this part, participants are tasked to evaluate the documentation of concrete ASAT rules. For each of the 12 rules used to validate our taxonomy in Section 3.2, we create a bespoke page in the survey. The page includes a screenshot of the rule’s documentation, a link to its official documentation, and a series of questions regarding its quality. As our goal is not to evaluate the ability of participants to annotate the rule with our taxonomy, the screenshot includes information regarding its purposes and content types (as shown in Figure 2), agreed upon by four authors (the screenshots annotated are available in our replication kit).11 The first question asks whether the participant already knows the rule. Then, for each purpose and each content type, a question...

---

8https://icpc2024-asats.github.io?page=survey
9https://docs.rubocop.org/rubocop/cops_style.html#stylefetchenvvar
Table 4: Our questionnaire survey’s questions. We used linter instead of ASAT as it is more popular among developers.

<table>
<thead>
<tr>
<th>Question</th>
<th>Type</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer profile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is your experience as a developer?</td>
<td>Single choice</td>
<td>✓</td>
</tr>
<tr>
<td>Which of the following languages do you use regularly?</td>
<td>Multiple choices</td>
<td>✓</td>
</tr>
<tr>
<td>Do you know what a linter is?</td>
<td>Yes/No</td>
<td>✓</td>
</tr>
<tr>
<td>Do you use a linter on some of your projects?</td>
<td>Yes/No</td>
<td>✓</td>
</tr>
<tr>
<td>Which of the following linters were used in those projects?</td>
<td>Multiple choices</td>
<td>✓</td>
</tr>
<tr>
<td>Taxonomy evaluation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rate the usefulness of each purpose in the documentation of a linter</td>
<td>Single choice for each purpose</td>
<td>✓</td>
</tr>
<tr>
<td>For the What purpose, why do you think it is (not) important to be</td>
<td>Open-ended</td>
<td>✓</td>
</tr>
<tr>
<td>present in the documentation?</td>
<td>Open-ended</td>
<td>✓</td>
</tr>
<tr>
<td>For the Why purpose, why do you think it is (not) important to be present in the documentation?</td>
<td>Open-ended</td>
<td>✓</td>
</tr>
<tr>
<td>For the Fix purpose, why do you think it is (not) important to be present in the documentation?</td>
<td>Open-ended</td>
<td>✓</td>
</tr>
<tr>
<td>Do you think that there are other purposes that a linter documentation should have?</td>
<td>Open-ended</td>
<td>✓</td>
</tr>
<tr>
<td>Rules analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Have you ever seen this rule?</td>
<td>Yes/No</td>
<td>✓</td>
</tr>
<tr>
<td>For the rule and taxonomy provided, evaluate for each type of content its importance to explain the What purpose</td>
<td>Single choice for each type</td>
<td>✓</td>
</tr>
<tr>
<td>For the rule and taxonomy provided, evaluate for each type of content its importance to explain the Why purpose</td>
<td>Single choice for each type</td>
<td>✓</td>
</tr>
<tr>
<td>For the rule and taxonomy provided, evaluate for each type of content its importance to explain the Fix purpose</td>
<td>Single choice for each type</td>
<td>✓</td>
</tr>
<tr>
<td>General feedback</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Please comment freely on the linters documentation you saw: what you appreciated, disliked, and how it compared with your expectations.</td>
<td>Open</td>
<td>✓</td>
</tr>
</tbody>
</table>

We ask to evaluate the importance of this content type to document this specific purpose using the asymmetric scale introduced above. Finally, the last series of questions asks the participant to judge the overall quality of the documentation for each purpose, using a symmetric scale to measure satisfaction: Very satisfied, Satisfied, Neither satisfied nor dissatisfied, Dissatisfied, Very dissatisfied. Participants may answer Not present to any question, indicating that there is no content of the appropriate type or that the rule does not document the purpose of interest. The Not present answer also serves as a quality check, as shown later in Section 4.2.

When participants complete the first two parts of the survey, one rule out of the 12 is drawn randomly and displayed. They can then opt-in to analyze another rule, drawn randomly from the remaining ones until there is no more rule to examine. When a participant evaluates all the rules or opts out, he is redirected to the last part of the survey.

General feedback. This final part consists of a single open-ended question asking the participants to comment on the documentation of the rules they evaluated. This question is designed to put our taxonomy aside and invite the participants to share their opinions more freely: what they liked and disliked about the rules and their documentation, and how it compared with their expectations.

Finally, we proceeded to a pilot testing of the survey with three software engineering researchers external to the study, who provided feedback about the phrasing of the questions and a time estimation to complete it.

4.2 Participants and Methodology

We primarily shared the survey with industrial partners and fellow researchers through direct contact and mailing lists. We also publicized it on social and professional networks (Twitter, LinkedIn, Slack). The survey was available online on a self-hosted LimeSurvey instance from July 4 to October 19, 2023. Overall, we received a total of 179 anonymous answers. The participants left at different stages: 179 entered their developer profile, 119 evaluated our taxonomy, 85 evaluated at least one rule (for a total of 289 rule evaluations), and 26 answered the last open-ended question.

Our methodology to sanitize the data and analyze the responses is as follows. First, we clean up the responses and attempt to remove noise. As mentioned earlier, the participants can answer Not present when a type of content or a given purpose is missing from the documentation of the rule they are evaluating. We observe that, in some cases, participants marked some type of content or purpose as Not present although it was present and marked as such in the screenshot. For instance, some participants answered that the Fix purpose for the Code was not present in the screenshot of Figure 2. In this case, as a sanity measure, we discard the participant’s answers related to this purpose for the given rule, for all content types. We apply the same filter when a participant provides an evaluation for a purpose that is not present in the rule presented to them. We obtained 225 evaluations for the What purpose, 91 for the Why purpose, and 161 for the Fix purpose.

Second, we use thematic analysis [28] to extract codes from the answers to open-ended questions. Two authors read these answers...
We only include the responses of the 85 participants that have a mean of 3 rules per participant. Each of the 12 rules has been evaluated at least one rule.

**Developer profile.** The distribution of participants in terms of experience is fairly even. Of the 85 participants, 37 have less than 5 years of experience as a developer (novices), 22 have between 5 and 9 years (juniors), and 26 have more than 10 years (seniors). The most used programming language is C/C++ (55%), closely followed by Python (54%), JavaScript and TypeScript (44%), and Java (42%). The remaining languages are used by less than 15% of respondents. A large majority (81%) of participants do know what an ASAT is; the proportion grows with experience (73% of novices, 82% of juniors, and 92% of seniors). The same trend is found for ASAT usage: 65% of the participants use or used ASATs in their projects (46% for novices, 68% for juniors, and 88% for seniors). There is also a noticeable imbalance in ASAT use depending on the programming languages they use: 53% of C/C++ developers use an ASAT, 61% of Python developers, 64% of Java developers, and 81% of JavaScript and TypeScript developers. A plausible explanation is that ESLint is often bundled by default when initializing JavaScript and TypeScript projects. When participants use ASATs in their projects, the most popular one is indeed ESLint (41%), followed by Pylint (21%), and SonarLint (19%). The remaining tools are used by less than 10% of participants; three respondents mention additional ASATs they use—clang tidy, Fortify and OCaml platform—indicating that our selection of ASATs reflects the ones used in practice. Overall ASAT usage reflects language use, with the exception of C/C++: since developers use more than one language, C/C++ users tend to use ASATs with other programming languages.

**Taxonomy evaluation.** In this part, we ask participants to judge the usefulness of the What, Why, and Fix purposes in the documentation of ASATs, independent of any particular rule. Figure 3a shows the results: the participants strongly expect each of the purposes to be present and documented. While the What and Why purposes are largely judged as essential, the Fix purpose appears slightly less essential to participants, but still worthwhile. In particular, we note the importance of the Why purpose to participants, which indicates that rule documentation should not only document the problem and its solution but also the rationale motivating the rule; in contrast, only half of the rules we analyzed had a rationale documented (Table 3). When grouping answers by developer experience, programming language, or other profile criteria we do not observe any major differences.

**Rules analysis.** The 85 participants analyzed a total of 289 rules, with a mean of 3.4 rules per participant. Each of the 12 rules has been evaluated by 19 to 29 different participants. Figure 3b shows how the participants rate the importance of each type of content to document each purpose in the rules they examined. If a rule does not include a particular type of content to document a given purpose, and the participant marks it as Not present, we omit this data point as it does not convey any positive or negative judgment. This explains why there is no evaluation of Code to document the Why purpose, as none of the 12 rules document it with code.

Participants evaluate the importance of the Why purpose positively (Essential or Worthwhile), regardless of its incarnation (94% for Text, 88% for Code, and 68% for Hyperlinks). While text and source code are seen as essential, the Hyperlinks are mainly judged as worthwhile. For the Why purpose, participants mostly evaluate Text as essential (69%) and Hyperlinks as worthwhile (68%). For the Fix purpose, participants mostly evaluate Text (59%) and Code (60%) as essential, and Hyperlinks as worthwhile (48%).

Participants evaluate Text and Code very positively to explain the three purposes. This suggests that a combination of text and source code might be the best choice to document ASAT rules.
Participants also evaluate Hyperlinks positively, with a minimum of 68% of positive evaluations for each purpose. Yet, most participants evaluate them as Worthwhile rather than Essential. Moreover, a significant portion judges them as Unimportant: three times or more than Text or Code—and more than they are judged Essential. A possible reason is that visiting external resources disrupts the reading flow and that important information may be lost among other resources, particularly if linking to larger documents. A solution could be to extract and distill the important information from external resources into the documentation, and cite it as a source.

Finding #3: Text and source code are best suited to document the What and Fix purposes, and good documentation for an ASAT rule should include both. Text is the best-suited medium to document the Why purpose. Hyperlinks are rarely seen as essential and should be used sparingly.

Finally, Figure 3c displays the satisfaction of participants regarding the quality of the documentation in the rules they evaluated, for each purpose. We observe that the participants are mostly satisfied with the quality of the documentation for the What (84%) and Fix purposes (70%). Yet, we observe that almost a quarter of the participants are not satisfied with the quality of the documentation for the Why purpose. Worse, in 13% of cases, participants are very dissatisfied with its quality. This strongly contrasts with the usefulness evaluation emitted by participants in Figure 3a.

Finding #4: Participants express a strong interest in understanding the rationale behind ASAT rules when reading their documentation (Why). Yet, Finding #1 indicates that only 50% of ASAT rules document the Why purpose; when present, participants are dissatisfied with how it is documented in close to 25% of the cases. Clearly, documents should consistently include and explain the Why aspect.

4.4 Qualitative Analysis

In the following, we summarize free-form comments based on our coding. Codes mentioned for the first time include their frequency like this (0). Code referenced after being introduced once is like this. We do not report on “obvious” codes (e.g., understanding the rationale (29) for the why purpose).

Transversal theme: learning. One of the most salient themes, across all three purposes (learning-what (11), learning-fix (13), and particularly learning-why (21)), is learning: 45 comments touched on that theme in one way or another. This is sometimes phrased as self-improvement of the developer’s skills, particularly for beginners or (more occasionally) for onboarding new team members. Comments on the What aspect mention the need to understand the error to not repeat it (thus improving skills), as well as team aspects (“because it facilitates the integration of new members”). Comments on the Why are the most prevalent. Explaining the rationale for an error is important to understand its purpose and importance, and from that, remembering it: “If I don’t know why, then I don’t know why it’s a bad thing and I cannot improve as a developer”. Finally, the Fix is more immediate. Once the problem is known and understood, learning of possible solutions is valuable: “It is likely the coder introduced a bad pattern/error due to lack of expertise; as such, it would be wrong to assume he/she will know how to fix it”.

Transversal theme: saving time. Respondents emphasized efficiency in all three aspects (saving-time-what (2), saving-time-why (3), particularly saving-time-fix (13), automated fixes (4)). For instance, one respondent wants to “understand in seconds what a lint is about”, which requires clear and concise explanations. Missing information in the Why prevents one from deciding whether to act on a warning (“I will probably lose time looking it up on the internet. Also I will be less motivated to fix it”). Since fixes are the most actionable, there is more demand to have standard solutions available to solve the issue quickly, rather than searching for information on the web or asking teammates. Having to search for the information is perceived to increase the chance a warning is ignored. Going further, the logical end is automation: “ideal thing is to just click a button to ‘autofix’ the issue, when available”.

Aspects specific to the What. The What provides understanding of what triggers a rule. It should be written clearly and concisely to this as easy as possible (thus saving time). The What helps finding where the warning is (8), which is not always obvious as “many different things may be discussed/considered on a single piece of code”. A key step in finding where the error is, is to relate the error to one’s code (12): rules are either described in the abstract, or, at best, with an example (3). Examples are preferred for this: “simpler the example [sic], the easier it is to relate to one’s code”. Another theme relates to false positives and negatives (5). Rules are implemented by heuristics that may be imperfect especially for complex cases (e.g. regular expressions). Describing what triggers a rule in details is useful to disambiguate between true positives and false positives, as well as knowing cases that the rule can miss (false negatives).

Aspects specific to the Why. The Why is key to deciding whether to act on the warning, or not. Developers analyse the tradeoffs and risks involved (11): the Why should in particular explain the severity of the warning: “I can judge whether the criticality justify modifying this piece of code. I may choose to disregard the rule if I judge it not worthwhile”. In some cases, the rule’s relevance (9) will be questioned (such as when it is a false positives and negatives, or when it is a matter of personal or team preferences (5), rather than a real issue. Thus the Why should motivate and justify the effort (10) that will be invested in fixing the warning; needless to say, if said effort is low (saving time via good examples or automated fixes), it will be easier to act on it.

Aspects related to the Fix. There is more debate as to the importance of the Fix, compared to the What and the Why. Some respondents state fixes are less or not important (7) for several reasons: either because the rules are simple, the fixes would be too basic, or because the Why and What are sufficient (“in most cases previous information should be enough to infer how to fix”). For other respondents, fixes are essential or very important (10). For them, documentation without a fix is not actionable: “If the Fix is not here, understanding the what and the why lead us to nowhere”. Fixes increase ASAT friendliness (5), and help saving time. Fixes...
are particularly useful when they are not obvious (4), either due to lack of knowledge (learning) or due to their difficulty. Providing examples (10) is useful to clarify complex rules. Some respondents mention that fixes may not be optimal (7) given the context, at worse they can "lead beginners to apply a cascade of bad decisions made to satisfy the linter".

Other purposes for documentation. Most respondents mentioned no additional purpose (14) than the What, Why, and Fix. A few respondents did mention other documentation purposes, such as exceptions, alternatives, and limitations (3), risks (3), or configuration (2) of the ASAT warnings. Indeed, we encountered these elements and included them in our nomenclature, but decided to exclude them from the remainder of our analysis as they dealt with more operational aspects.

Additional comments on the documentation. Several participants emphasized in their free-form comments some topics that emerged earlier. Several respondents highlighted the need for a summary (7) ("Sometimes too much text which does not encourage taking the time to read and therefore deal with the error" or a structured template (6) where "the what, the why and the fix are clearly separated"). Others reiterated the need to use code examples (4), including adding examples of both compliant and non-compliant code, or to avoid links (2) ("external links are almost never useful").

Finding #5: ASAT documentation has important learning purposes (particularly the Why). Respondents read the What to understand the error, the Why to decide whether to address the warning, and the Fix to remove it. Missing elements make fixes less likely. Some respondents even suggested the use of a structured template to enforce the presence of every important purpose. Additionally, some respondents expressed concern about efficiency. A summary was suggested by some respondents to speed up the understanding of the rule. Respondents pleaded the use of code examples and recommended to use external links sparingly to avoid breaking the reading flow.

5 THREATS TO VALIDITY

External validity. Our study bears several threats with regard to external validity. The first threat is that our corpus of ASATs and ASATs rules documentation is not representative, and is even biased toward popular ASATs. Therefore, we have no guarantee that our results would generalize to the actual population, especially the percentages displayed in Table 2 and Table 3. Another threat is that the respondents of our survey are not a random sample of the population of ASATs users. As a consequence, the results we obtained with our participants might not generalize, especially the percentages displayed in Figures 3a to 3c.

Internal validity. We extensively use qualitative methods to build our nomenclature and taxonomy (open card sorting [23]) and analyze the open answers to the survey (thematic analysis [28]). It is well known that these methods can be affected by subjectivity [29]. As a consequence, different researchers might have obtained a different nomenclature, taxonomy, and other themes from the survey answers. As a mitigation measure for the taxonomy, we performed an internal validation and a reality check in the survey where we observed that it was well understood by the respondents. For the other results obtained from qualitative analysis, we systematically used harmonization sessions to reduce the subjectivity of our findings.

With regard to the answers of the participants to the survey, there is a chance that they did not fully understand the questions we asked. This threat affects mostly the closed-ended questions where we cannot do any sanity check by looking at the answer. The most difficult questions concerned the analysis of the usefulness of each type of content to document each purpose, as the participants needed to carefully analyze how we rated the content of the rules. As a sanity check, we included a specific scale item (Not present) to double-check that the participants did understand our rating. We used this sanity check to filter out incoherent answers. However, this sanity check is not perfect and it is possible that participants answered these questions without understanding our rating.

6 RELATED WORK

To the best of our knowledge, there is no prior work having studied the rules documentation of ASATs. In the remainder of the section, we discuss the related work according to two topics: studies of ASATs and studies of software documentation.

6.1 ASATs Studies

Researchers have emphasized various benefits associated with the use of ASATs. Tómasdóttir et al. highlight eight such advantages, such as error prevention, keeping the code simple and consistent, or improving the efficiency of code review and discussions [14]. ASATs can also automate compliance to a standard coding style [30].

Several studies explore the reasons inhibiting the use of ASATs by developers. Tómasdóttir et al. highlight that the agreement on rules to enable, especially for pre-existing projects, is an important issue and makes it difficult for developers to follow the rules when they disagree [14]. Other challenges were also raised: dealing with false positives [31], having better integration into the development process [32] and missing quick fixes when rules are detected [18].

The study from Novak et al. [1] produces a taxonomy of ASATs across 10 categories (such as rule domains or configurability). However, ASAT documentation has not been investigated in their study.

The studies on the warnings (also called notifications) content of ASATs are closely related to ours, as warnings are the first piece of documentation that developers see when using an ASAT. An important result is that their content might not be sufficiently helpful to developers [7, 17, 18]. Another finding is the need to have clear and concise warnings, especially when developers use ASATs in CLI [33]. It aligns with our findings for ASAT rule documentation: the main information should be quickly available and understandable. Buckers et al. present a tool and its evaluation to help developers treat more efficiently the numerous notifications and their contents [34]. Overall, our results are consistent with those of previous related studies on warnings [7, 15, 17, 18, 32].

Tahaei et al. find that examples are more valued and links less valued, which echoes our study [7]. They focus on four security warnings, while we cover more domains beyond security and explore more rules and ASAT tools.
Johnson et al. study notifications for an ASAT, a coverage tool, and a compiler (five examples each) [17]. Since they focus on multiple tool types, ASAT-specific aspects (e.g., examples) are less prominent. However, they identify "Problem Importance Gaps" and "Problem Resolution Gaps" that echo our Why and Fix purposes.

Nachtgall et al. define the objective of warning messages as such: "Warning messages [...] have to direct the developer's attention to the detected issue and give information on what might be wrong, why it should be fixed, and how it could be fixed", echoing our taxonomy. However, it is not clear how they came up with this definition, while we obtain it from an analysis of existing ASAT documentation and confront it to the expectations of developers. [18]

Do et al. extract three features (among many others) expected by ASAT users from a tool and a literature review (called F3, F4, and F7 in their article), which are similar to explaining the What, Why, and Fix purposes. These three features are among the most important ones according to the developers they surveyed [15]. The other features are more tool-oriented since their study focuses on tools and not documentation.

Finally, Johnson et al. make the following observation: "Nineteen of our 20 participants, felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem, and what they should be doing differently" which also echoes our taxonomy [32].

One important novel aspect of our study is that it is the only one that focuses on reference documentation, not warnings or tools. It confirms that the findings of the study about warnings are transferable to reference documentation. In addition, our results provide more evidence about what is present/missing from existing documentation both for the nomenclature and taxonomy, while the qualitative part dives deeper into these topics.

### 6.2 Software Documentation

Another related topic is the study of documentation of Application Programming Interfaces (APIs). API documentation shares a common goal with ASAT documentation as they are meant to teach developers how to correctly use an API as quickly as possible [19]. On the other hand, they focus on a single software component while ASAT documentation describes rules that apply in a wide range of contexts. We also find similar expected criteria with the writing of the rules documentation, as the need to include short code snippets demonstrating API usage in context or code examples illustrating the best practices [35, 36].

Aghajani et al. performed an empirical study on software documentation resulting in a taxonomy of documentation issues [37]. An interesting outcome is the identification of themes relatively close to ours, especially Information Content similar to our Comprehension. They pursue their work in another article [38] by realizing two surveys, first to evaluate the relevance of issues they had found, then to explore the needs on type of documentation. It results in guidelines to improve the state of the art around documentation.

Regarding software documentation in general, one major flaw in software documentation is the difficulty of keeping it updated through its lifetime [39]. Some developers simply assume it is outdated and do not rely on it [40]. At first glance, we did not see this problem in the documentation of ASAT rules maybe because they evolve less often, but it would be an interesting issue to study.

### 7 CONCLUSION

In this article, we explored how ASATs rules are documented, and contrasted them with developers’ expectations through several contributions. We first studied how more than 100 rules—spanning 16 ASATs in multiple programming languages—are documented, leading to a nomenclature of documentation elements refined in a taxonomy of documentation purposes and content types. We then use this taxonomy to contrast the documentation of 12 real-world rules with developer expectations via a survey involving 85 respondents who evaluated the rules 289 times. Among other findings, we highlight issues with the Why purpose: despite being considered essential by developers to decide whether to act on a warning, half of the rules miss it; of the rest, a quarter of survey responses point at quality issues. We also find out that code examples in addition to text are attractive for documenting the What and Fix purposes. Finally, some developers expressed concern about saving time, leading to the recommendation to include summaries, and reducing the use of external links that disrupt the reading flow. In future work, we plan to extend our study to more ASATs and to more quality issues inside the documentation, such as outdatedness, as has been done for API documentation [37, 38].

### ACKNOWLEDGMENTS

We would like to thank Matias Martinez, Lima Ochoa Venegas, and Théo Zimmermann for their initial feedback on the online survey.

### REFERENCES


