DocPrism: Multi-lingual Detection of Incorrectness Inconsistencies between Code and Documentation

XIAOMENG XU, University of British Columbia, Canada
ZAHIN WAHAB, University of British Columbia, Canada
REID HOLMES, University of British Columbia, Canada
CAROLINE LEMIEUX, University of British Columbia, Canada

Code-documentation inconsistencies are common and undesirable: they can lead to developer misunderstandings and software defects. This paper introduces DocPrism, a lightweight multi-language, code-documentation inconsistency detection tool. DocPrism uses a standard large language model (LLM) to analyze and explain inconsistencies, and focuses on outputting *incorrectness* inconsistencies. Plain use of LLMs for this task yields unacceptably high inconsistency flag rates—i.e., over 90% of functions are flagged as inconsistent with their documentation. One substantial reason is that LLMs identify natural gaps between high-level documentation and code as *incompleteness* inconsistencies. We introduce and apply the *Local Categorization, External Filtering* (LCEF) methodology: LCEF uses an LLM's local completion skills, rather than its long-term reasoning skills, to focus on reporting *incorrectness* inconsistencies. In our ablation study, LCEF reduces DocPrism's inconsistency flag rate from 98% to 14%, and increases F1 score from 0.22 to 0.77, compared to standard prompting techniques. On a broad evaluation across Python, TypeScript, C++, and Java, DocPrism maintains a low flag rate of 17%, and achieves a precision of 0.63 without performing any fine-tuning. We also establish a conservative lower bound across four programming languages, showing that inconsistency errors are present in 11% of code-documentation pairs. In addition, DocPrism achieves precision comparable to the state-of-the-art on an established synthetic dataset, but substantially outperforms it on our real-world Java dataset in precision (DocPrism: 0.47–0.67 vs. SOTA: 0.05–0.14).

CCS Concepts: • **Software and its engineering** → **Documentation**; *Software maintenance tools.*

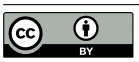
Additional Key Words and Phrases: code-documentation inconsistency, documentation, code comments, LLM

ACM Reference Format:
Xiaomeng Xu, Zahin Wahab, Reid Holmes, and Caroline Lemieux. 2026. DocPrism: Multi-lingual Detection of Incorrectness Inconsistencies between Code and Documentation. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA157 (October 2026), 22 pages. https://doi.org/10.1145/3832248

1 Introduction

Method-level documentation acts as a specification describing the behaviours a method provides. Documentation serves as an abstraction developers use to understand how to correctly use a method without having to read through its source code. Unfortunately, long experience has taught developers not to trust documentation, mainly due to inconsistencies between what the documentation says and what the code actually does. The goal of this paper is to *detect* and *explain* such code-documentation inconsistencies.

Authors' Contact Information: Xiaomeng Xu, University of British Columbia, Vancouver, Canada, xmxu@cs.ubc.ca; Zahin Wahab, University of British Columbia, Vancouver, Canada, zwb@cs.ubc.ca; Reid Holmes, University of British Columbia, Vancouver, Canada, rtholmes@cs.ubc.ca; Caroline Lemieux, University of British Columbia, Vancouver, Canada, clemieux@cs.ubc.ca.



ACM 2994-970X/2026/10-ARTISSTA157
https://doi.org/10.1145/3832248

This work focuses on surfacing *incorrectness* inconsistencies between code and documentation. First, cases where there is a logical conflict between code and its documentation: we call this a *direct mismatch*. Second, cases where a behaviour is documented but not implemented: we call this an *over-promise*. Both of these are true incorrectness inconsistencies: they imply that either the code or its documentation is erroneous.

Identifying inconsistencies between documentation and code is hard to do, as documentation is written with natural language and code is written with programming languages. Rather than considering a limited set of inconsistency patterns in a single programming language [21, 22, 25, 27], we seek to *explore the range* of developer-written inconsistencies that exist in practice, across multiple programming languages. Rather than treating inconsistency detection as a binary classification problem [3, 6, 7, 9, 13, 14, 17, 20, 26], we seek to (1) *find* conflicting fragments in the code and/or documentation and (2) explain the inconsistencies between them. Given that it is often difficult to determine whether the code or the documentation is buggy, we believe this exposition could help inform developers of potential inconsistencies and promote further fixes.

Pre-trained LLMs enable generic code-documentation inconsistency detection. Unfortunately, we found that when asked to identify code-documentation inconsistencies under standard prompting techniques, LLMs have unacceptably high flag rates (i.e., the number of tested functions flagged as inconsistent). Not only does the open-source model LLaMA3.1-70B achieve a flag rate of 90-97%, but the proprietary model GPT4.1 also reaches 82-91% on our in-the-wild datasets of 1,991 functions from 20 real-world codebases across Python, TypeScript, C++, and Java. Such high flag rates are undesirable, as they could lead to alert fatigue and the abandonment of the analysis tool [18]. Moreover, we observe that a substantial proportion of the identified inconsistencies arise from LLMs eagerly pointing out that some source code details are undocumented. We call such inconsistencies *under-promises*. Under-promises are *incompleteness* inconsistencies that reflect the natural tendency of documentation to be more concise than code. Surfacing all of these would not only obscure the *incorrectness* issues we seek to surface, but also result in a flag rate so high (>90%) that it becomes untractable for human analysis.

We introduce DocPrism, a lightweight, multi-language code-documentation inconsistency detector, with the integration of our *Local Categorization and External Filtering* (LCEF) methodology. LCEF guides LLMs to categorize different types of inconsistencies, with the goal of surfacing incorrectness inconsistencies. *Local Categorization* turns the inconsistency detection into a local completion problem, rather than a long-term reasoning problem, and leverages the chain-of-thought approach to reduce spurious outputs. In addition, DocPrism includes a front-end that highlights the detected conflicting fragments in the code and/or documentation for each tested code-documentation pair, along with corresponding explanations.

In our evaluation, we run DocPrism on 1,991 developer-written code-documentation pairs *as they exist* in the wild. This allows us to evaluate DocPrism's real-world applicability. These functions are collected from 20 real-world open-source codebases across four programming languages: Python, TypeScript, C++, and Java. We find that DocPrism, using the LCEF methodology, is highly effective at surfacing incorrectness inconsistencies: over 96% of surfaced inconsistencies are incorrectness ones. DocPrism also greatly reduces the number of functions flagged inconsistent with their documentation: decreasing the flag rate from 90-97% to 14-19%. This makes our tool's output much more tractable for developer usage.

Though DocPrism's goal is to surface incorrectness inconsistencies, our ablation study shows it is effective at increasing the overall precision, from 0.14 to 0.71. Due to the well-documented noisiness [13, 17, 26] in the labels of existing automatically labelled dataset [13], we manually label the results in our datasets. Through our manual labelling, we categorize DocPrism's correctly and incorrectly identified inconsistencies. This manual analysis also allowed us to confirm the

presence of incorrectness inconsistencies in real-world software systems, even those that are popular and well-maintained. As such, our evaluation provides a conservative lower bound on code-documentation inconsistencies in practice.

In addition, we compare DocPrism with C4RLLaMA, a state-of-the-art tool. C4RLLaMA outperforms prior work on the Panthaplackel dataset [13], which is constructed from synthetically paired (cross-commit) code and documentation. On this dataset, DocPrism achieves a precision of 0.78 while C4RLLaMA achieves a precision of 0.83. On our Java dataset, which we refer to as the DocPrism dataset, DocPrism's precision drops slightly from 0.78 to 0.58. In contrast, C4RLLaMA's precision drops substantially from 0.83 to 0.08. This suggests a distribution shift between their dataset and ours, which contains code-documentation pairs that occur together in the same commit.

In short, our contributions are as follows:

- The Local Categorization-External Filtering (LCEF) approach to reliably surface *incorrectness* inconsistencies, while maintaining a reasonable (i.e., significantly below 100%) flag rate.
- The DocPrism tool, a lightweight, multi-language inconsistency detector that implements LCEF and a frontend to surface generic inconsistencies. DocPrism highlights conflicting code/documentation snippets along with explanations.
- We observe different performance by the state-of-the-art, C4RLLaMA [17], on the DocPrism dataset versus the Panthaplackel dataset [13]. This suggests future work evaluating on such synthetically created inconsistencies should also evaluate on naturally occurring ones in real-world codebases.
- A conservative lower bound on incorrectness issues between functions and their docs. Through our manual validation of DocPrism results, we found incorrectness inconsistencies in 11% of examined functions, across 20 projects in 4 programming languages.

2 Related Work and Design Decisions

We begin by contextualizing how current literature motivates our design and evaluation decisions.

2.1 Generic vs Specific Inconsistency Detection

In building a code-documentation inconsistency detection tool, a decision must be made about what kinds of inconsistencies should be detected. Some approaches choose to focus only on a few types of inconsistency which can be mapped to particular natural language patterns (*pattern-based*). Others attempt to use more sophisticated natural language analysis to surface *generic* inconsistencies.

Pattern-Based Inconsistency Detection. As documentation is a natural language artifact, handling it in a precise manner requires careful use of NLP methods. One solution is to focus on analyzing a few natural language patterns whose meaning can be precisely understood. icomment [21] looks for rule-containing documentation by checking for imperative words (e.g., *should*, *must*), and filters this down to lock-related and call-related rules with other keywords (e.g., *lock*, *acquire*, *release*). It achieves a precision of 0.61 on a C/C++ dataset. tcomment [22] uses the structure of javadocs to infer rules about whether specific parameters are nullable, and checks these inferred rules against dynamic behaviour. It achieves a precision of 0.48 on its Java dataset.

Even with LLMs, tools may focus on particular patterns of inconsistencies. MPChecker [25] focuses on *multi-parameter constraint* inconsistencies between API documentation and code. The approach relies on symbolic execution to extract code inconsistencies, making it hard to apply beyond its Python context. RustC4 [27] uses LLMs to extract an integer range, collection emptiness, and index constraints from documentation. It compares these to constraints extracted by a static analysis pass over Rust ASTs. Both of these approaches achieve very high precision (> 0.95), but remain restricted to particular patterns of inconsistencies in particular programming languages.

Generic Inconsistency Detection. A number of works use machine learning to identify arbitrary code-doc inconsistencies. In theory, they could find any semantic difference between code and its documentation. Most learning-based approaches [3, 6, 7, 9, 13, 14, 17, 20, 26] treat inconsistency detection between code and comments as a binary classification problem. They train [3, 6, 7, 9, 13, 14, 20] or fine-tune [17] machine learning models to achieve better performance across metrics including accuracy and F1 score.

To the best of our knowledge, C4RLLaMA [17] is the state-of-the-art (SOTA) in both *just-in-time* and *post-hoc* inconsistency detection, outperforming relevant prior works on a publicly available Java dataset [13]. C4RLLaMA is a fine-tuned LLM based on CodeLLaMA that performs binary inconsistency detection between code and documentation. It also provides rectifications (i.e., fixed versions of the documentation) when the binary label is inconsistent.

Design Choice: Our goal is to detect inconsistency errors developers introduce into real-world software systems; we thus focus on *generic* inconsistency detection. Our aim is similar to prior generic detection approaches [3, 6, 7, 9, 13, 14, 17, 20, 26], in that we also rely on machine learning manage the gap between natural language and source code. This allows us to find generic inconsistencies, but trades-off precision for generality compared to the pattern-based approaches. Unlike most of prior work, we do not rely on a pre-labelled dataset, but instead use manual labelling to evaluate our tool.

2.2 Timing of Inconsistency Detection

Inconsistency detection between code and documentation can be conducted at two different cadences: *Just-In-Time* and *Post Hoc* [13]. *Just-In-Time* means the tool is run when the code or documentation are being modified. This provides extra input to a detection approach beyond the code and its documentation: the *diff* describing how the code or documentation has changed. *Post Hoc* approaches, on the other hand, perform inconsistency detection at some other time, lacking information about a specific *diff* or commit.

Design Choice: To maximize flexibility, we want DocPrism to apply in as many development situations as possible. By supporting *Post Hoc* execution, DocPrism does not need to be tied to a specific *diff*. This means DocPrism can analyze code repositories in the wild, where inconsistencies may have been committed, without having to look through commit history. Thus DocPrism is not comparable to the *Just-In-Time* approaches [6, 10, 19, 20, 26] which require some representation of the *diff* to work. Orthogonally, most *Just-In-Time* approaches are built and evaluated for Java [10, 19, 20, 26], with one approach also evaluated on Python [6]. We evaluate DocPrism on Python, TypeScript, Java, and C++.

2.3 Surfacing Detected Inconsistencies

What does it mean to detect an inconsistency? Many prior works on generic inconsistency detection treat inconsistency detection as a binary classification problem [3, 6, 7, 9, 13, 14, 20, 26]. Thus, the only thing these tools can surface to developers is a binary label: *"this function is inconsistent with its documentation"*. A single binary label may be sufficient for a developer analyzing a shorter method, such as the one in Fig. 2. However, once methods are longer, it is much harder to pinpoint, from a single binary judgment, what inconsistency has been detected. We aim to provide more information to developers, so they can quickly check whether the detected inconsistency is valid or not.

Some approaches aim to both classify and *automatically rectify* inconsistencies in the tested code or comments [10, 17]. This gives much more information to the user about the detected inconsistency. If the user compares the original documentation and the rectified documentation, they can try to back-infer the detected inconsistency. But automated documentation rectification

assumes that documentation is always the incorrect artifact. We argue that an incorrectness inconsistency likely indicates an error, but we *cannot* know if it is an error in the *documentation* or the *code*. Consider Fig. 2: without detailed knowledge about the codebase, we cannot know whether the code is missing an addition, or the documentation is an incorrect specification.

Design Choice: Because we cannot know whether code or documentation should be rectified, and current tooling is not accurate enough to fully automate rectification [10, 17], we believe it is essential to keep developers in the loop. Instead of automatically fixing inconsistencies, DocPrism *exposes and explains* them. Given a function and its documentation, DocPrism outputs an HTML page which highlights inconsistent code and documentation snippets and provides an explanation. Fig. 1, 2 and 3 are all examples of DocPrism output. We believe this output format allows developers to be clearly informed about the nature of the inconsistency so they can fix the code or documentation accordingly. This output also improves our own ability to accurately manually label any given result.

2.4 Data Labelling and Contamination

Manually labelling inconsistencies is hard. But training machine learning models requires plentiful labelled data. Several recent approaches [3, 17, 26] use Panthaplackel et. al's [13] dataset. The procedure to collect this data takes two consecutive versions of a (documentation, code) pair: $\langle D_1, C_1 \rangle$ and $\langle D_2, C_2 \rangle$. Then, it labels the pair $\langle D_1, C_2 \rangle$ as *inconsistent* if the code *and* documentation changed between versions (i.e., $D_1 \neq D_2, C_1 \neq C_2$), and labels the pair $\langle D_1, C_2 \rangle$ as *consistent* if only the code changes (i.e., $D_1 = D_2, C_1 \neq C_2$). This falsely labels pairs as consistent when code changes in a semantically meaningful way, but documentation fails to be updated. Further, it has been pointed out that if documentation changes in a semantics-preserving manner, $\langle D_1, C_2 \rangle$ will be falsely labelled as inconsistent [26]. Finally, the entire set of inconsistencies are doc-code pairs which never actually occur together in codebases ($\langle D_1, C_2 \rangle$ does not occur in the codebase if $D_1 \neq D_2$). This introduces potential distribution shift between these synthetic datasets and code in the wild.

The labels in this dataset are known to be noisy. Panthaplackel et. al included a "clean" subset, where these labels are validated by humans [13]. Post Hoc performance on the clean dataset was slightly lower than on the full dataset, with precision in the $0.54 - 0.61$ range rather than the $0.56 - 0.63$ range. This clean dataset has been reused in some subsequent work [26], but not all [3, 17]. Recent works included training methodologies to account for noise in the dataset labels [17, 26]. As part of creating this cleaned dataset, Panthaplackel et al. [13] found that labels were wrong for 26-28% of *inconsistent* examples, and 8-12% of *consistent* examples. If the sample analyzed for the clean dataset creation is representative of the rest of the sample, this is a large amount of noise.

There is an additional threat with any procedure for collecting labelled data which relies on historical information. As LLMs are trained on a huge amount of code, if we rely on historically rectified documentation issues to create a dataset, this may lead to data contamination between the LLM's training set and our test set.

Design Choice: Due to these shortcomings, we choose to run our tool on code and documentation as it exists in-the-wild. In particular, we curate functions (or methods) with existing documentation, and run our tool on these pairs. We believe this makes our evaluation less vulnerable to dataset contamination, as this should be the final state of documentation in the LLM's training set. However, it means we need to manually analyze the results of our tool to label them as correct or not.

Manual labelling allows us to categorize reasons for our approach's strengths and weaknesses. But, it means judgments are subject to human error, and inherently based on the opinion of one (for most of our Python, TypeScript, Java, and C++ datasets) or two (for our ablation dataset) computer scientists. Nevertheless, we think this evaluation remains representative of how DocPrism would

be used in practice: its outputs would be examined by a developer, who would then decide whether the inconsistency is relevant or not.

3 Code-Doc Inconsistency Categories

DocPrism aims to identify code-documentation inconsistencies. That is, our focus is only on inconsistencies that can be spotted from analyzing the source code and documentation of a method. There may be issues in documentation beyond such inconsistencies, such as missing information about dependencies [1], or information about temporal relationships between methods [23]. These issues require integrating knowledge about a codebase *beyond* the source code of a method into documentation. Following prior work [1], we do not consider these a form of code-documentation inconsistency.

We separate code-documentation inconsistencies into three major categories: over-promises, direct mismatches, and under-promises. We aim to identify over-promises and direct mismatches, which are *incorrectness* inconsistencies. These are likely to indicate an error in either the documentation or the code. We believe these are higher priority to surface than under-promises, which are *incompleteness* inconsistencies.

3.1 Over-Promise

Over-promises are a type of code-documentation inconsistency which point to some part of the documentation not being implemented. These correspond to the *"Behaviour described in documentation but not implemented"* subcategory of code-documentation inconsistency described by Aghajani et al. [1]. Fig. 1 shows a typical example of an *over-promise* found by DocPrism in the Python pandas project.[1] The documentation states that the function should "keep track of whether numexpr was used by storing 'True' for each successful use of evaluate with numexpr". However, this feature is not implemented. Over-promises are *incorrectness* issues because functionality developers expect based on the documentation is not present in the implementation.

```
def set_test_mode(v: bool = True) -> None:
    """Keeps track of whether numexpr was used. Stores an additional 'True' for every successful use of
    evaluate with numexpr since the last 'get_test_result'. """
    global _TEST_MODE, _TEST_RESULT
    _TEST_MODE = v
    _TEST_RESULT = []
```

Explanation generated by DocPrism: *"The code does not implement storing a True value for every successful use of evaluate with numexpr. It only sets test mode and resets test result."*

Fig. 1. A Python over-promise identified by DocPrism, along with its highlighting and rationale (`Pandas-48`).

3.2 Direct Mismatch

Direct mismatches are code-documentation inconsistencies which point to some part of the documentation logically conflicting with some part of the code. These are akin to the *"Code must change to match doc"* subcategory of code-documentation inconsistency described by Aghajani et al. [1]. However, we do not make a value judgment as to whether it is the code or documentation that must be changed in this case. Fig. 2 shows an example of a *direct mismatch* identified by DocPrism in the TypeScript Hoppscotch project.[2] The documentation states that the return value should be "the count returned is highest `Order-Index` plus 1". However, the code simply returns `rootCollectionCount[0].orderIndex` without adding one to it. These mismatches are

[1]Examples are identified as `codebase-n`, corresponding to our n^{th} function from the codebase `codebase` in our artifact.
[2]https://github.com/hoppscotch/hoppscotch

incorrectness issues, because there is a contradiction between what the implementation does and what its documentation says it will do.

```
/**
 * Returns the count of root collections present
      for a given teamID.
      The count returned is highest OrderIndex + 1
 * @param teamID The Team ID
 * @returns Number of Root Collections
 */
private async getRootCollectionsCount(teamID:
      string) {
    const rootCollectionCount =
      await this.prisma.teamCollection.findMany({
        where: { teamID, parentID: null },
        orderBy: { orderIndex: 'desc' },
      });
    if (!rootCollectionCount.length) return 0;
    return rootCollectionCount[0].orderIndex;
}
```

Explanation generated by DocPrism: *"The code returns the orderIndex as is, without adding 1 to it."*

Fig. 2. A TypeScript DocPrism-identified direct mismatch, including highlighting and rationale (Hoppscotch-37).

```
/**
 * @name SORTED_FLOATS::operator[]
 * Return the floating point value of the given index
      into the list.
 */
float SORTED_FLOATS::operator[](int32_t index) {
   it.move_to_first();
  return it.data_relative(index)->entry;
}
```

Explanation generated by DocPrism: *"The documentation does not mention moving to the first element before accessing the data."*

Fig. 3. A C++ under-promise identified by DocPrism, including highlighting and rationale (Tesseract-89).

3.3 Under-Promise

Finally, under-promises inconsistencies point to some part of the code not being documented. These do not correspond to any type of code-documentation inconsistency in Aghajani et al. [1]; i.e., Aghajani et al. do not characterize these as inconsistencies at all. Fig. 3 shows an example of an under-promise in the C++ Tesseract project.[3] This function's documentation does not describe that an iterator is moved to the beginning of the list before returning the value at a specific index in the list. We argue that this is an implementation detail a developer using this method would not need to know. So, it is not an incorrectness issue. Depending on a project's documentation practices, some under-promises may be viewed as *incompleteness* issues. But because they are not *incorrectness* issues, DocPrism aims to *not* surface under-promises.

4 DocPrism

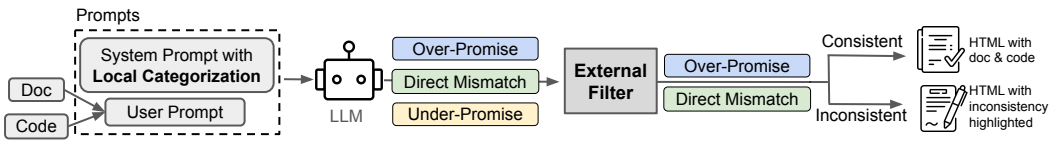


Fig. 4. DocPrism's workflow for detecting incorrectness code-doc inconsistencies.

We now describe DocPrism. As shown in Fig. 4, given as input a function and its documentation, DocPrism first builds system and user prompts, in which it applies our *local categorization* approach (§ 4.1). Second, it feeds these prompts to the LLM for inconsistency detection (§ 4.2). Our prompt design leads the LLM to implicitly categorize inconsistencies into the three categories from Section 3, and output them in JSON. Third, DocPrism performs *external filtering* to remove *under-promises*, and generates an HTML page reporting inconsistencies in the other two categories (§ 4.3).

[3] https://github.com/tesseract-ocr/tesseract

4.1 Prompts

Our prompts are brief by design. The system prompt consists of two parts: a prefix that introduces the task and input format, and a JSON schema that specifies the output format. The user prompt contains a pair of a function and its method-level documentation. As such, the LLM performs the inconsistency detection task in a zero-shot manner.

4.1.1 System Prompt - Prefix. The prefix defines the LLM's role, specifies the task, and indicates the input format. The full prefix is as follows:

```
You are a code review expert for the {project_name}{"library"/"framework"/"project"}, working on identifying
any inconsistencies between function-level code and its documentation. Given the code (enclosed within
[CODE] [/CODE]) and its accompanying documentation (enclosed within [DOCUMENTATION][/DOCUMENTATION]), please
identify any inconsistencies or information mismatches between them.
```

4.1.2 System Prompt - JSON Schema. We break down the task of code-documentation inconsistency detection into five subtasks, similar to a chain-of-thought approach [24]. The first two subtasks require the LLM to generate summaries of (1) the input documentation and (2) the code. These tasks aim to prime the LLM's understanding of the code and documentation being analyzed. The final three subtasks instruct the LLM to identify inconsistencies in each category (ref. Section 3): (3) *over-promise*, (4) *direct mismatch*, and (5) *under-promise*. Instead of explicitly stating the five reasoning subtasks in the prompt, we encode them into a predefined JSON schema with eight keys.

Summary Subtasks. *(Subtasks 1-2).* The first and second JSON keys, `"Documentation_Summary":...` and `"Code_Summary":...`, prompt the LLM to summarize the documentation and the code. We include the word `Summary` in both keys to emphasize brevity, aiming to reduce hallucinated content.

Local Categorization Subtasks. The remaining three pairs of JSON keys apply the approach of *local categorization*, which *locally* and *implicitly* guides the LLM to identify over-promises, direct mismatches, and under-promises. Each pair has a check-in key (**CK**) and a follow-up key (**FK**). The check-in key is a yes/no question tailored to detecting inconsistencies in each category. If the LLM's answer to the check-in-key indicates the presence of inconsistencies in the category, the follow-up key prompts the LLM to explain these inconsistencies.

In this manner, the task of identifying inconsistencies between code and documentation, and the category of these inconsistencies, is reduced to a *local completion* problem. The LLM needs not "remember" the definitions of each inconsistency category; it needs only complete the output format. As the LLM sequentially generates outputs for each pair of keys, it is implicitly guided to detect and list inconsistencies in each category, if any.

The next pair of JSON keys, *(Subtask 3)*, performs local categorization for *over-promises*. Then the next two keys, *(Subtask 4)*, perform local categorization for *direct mismatches*. For space reasons, we do not include the whole check-in and follow-up keys for these subtasks, but the full prompts can be found in our replication package.

(Subtask 5). The last pair of JSON keys prompts the LLM to identify *under-promises*. If the answer to the check-in key shown below is "Yes", the follow-up key asks it to output undocumented code snippets with explanations.

```
(CK) → "Is_some_code_not_documented_or_mentioned_in_the_documentation?": "Yes" or "No"
(FK) → "If_yes_to_Is_some_code_not_documented_or_mentioned_in_the_documentation":
       - "original_code_snippet_that_is_not_documented_or_mentioned_in_the_documentation"
       - "explanation"
```

The follow-up key is verbose; this is by design. In the early experiments, we found that this verbosity was necessary to ensure the LLM's outputs follow the JSON schema.

4.1.3 User Prompt. Since all detailed instructions are specified in the system prompt, we simply provide a clear test subject through the "user message". The following template is used for the user prompt, with placeholders for the documentation and code.

```
[DOCUMENTATION]{doc}[/DOCUMENTATION]\n [CODE]{code}[/CODE]
```

4.2 Inconsistency Detection

For each pair of a function and its documentation, DocPrism first generates the prompts as described above. To detect inconsistencies, it feeds these prompts to the open-source LLM LLaMA 3.1–70B [5]. The output is in JSON format, following the schema specified in the system prompt. In rare cases (3.7% of evaluated pairs) where the output does not match this format, we cannot extract inconsistencies from the LLM output. Thus DocPrism does not report any inconsistencies. We set the temperature to 0 to encourage deterministic results from the LLM.

4.3 External Filtering

DocPrism's External Filtering component takes in the JSON object output by the LLM. This component filters out inconsistencies in the third category, *under-promises*, by dropping content under the third check-in key/follow-up key pair (Subtask 5 in § 4.1.2). Our HTML backend reports any remaining inconsistencies by highlighting the relevant code/doc, and rendering the LLM-generated explanations. The DocPrism source code and all HTML files, including those for Figures 1-3 are included in the artifact (§ 9).

5 Experimental Methodology

Our evaluation seeks to provide insight into the kinds of inconsistencies present in real-world developer-written code-documentation pairs and the effectiveness of DocPrism at uncovering those inconsistencies. Experimental design decisions all have costs and benefits. Following best-practice advice [16], we report the **implications** of our design decisions along with our experimental methodology rather than a separate threats to validity section.

To perform this evaluation, we require access to real-world code and documentation written by developers. As discussed in Section 2.4, most recent prior work in this area relies on synthetic data that assumes all existing code-documentation pairs in existing code are consistent. The **implication** of this choice is that prior datasets do not align with developer beliefs that code-documentation inconsistencies exist in practice [8, 15, 21]. Consequently, we develop an authentic dataset consisting of real-world developer-written code-documentation pairs.

5.1 Evaluation Setup

All evaluations are conducted on an NVIDIA A100 80 GB GPU, the same hardware on which LLaMA 3.1–70B [5] performs inference. On average, DocPrism took approximately 22 seconds to evaluate each function. The **implication** of this choice is that our approach is primarily evaluated using a single model— although Section 6.4 examines whether proprietary model gives different results.

5.2 Evaluation Datasets

To perform our evaluation, we constructed two datasets. Our primary polylingual dataset is composed of code-documentation pairs from four programming languages and is used to gain insight into DocPrism and the nature of developer-written code-documentation inconsistencies. A second single-language ablation dataset is used to gain insight into specific DocPrism design decisions.

5.2.1 Primary polylingual dataset (Python, TypeScript, C++, and Java). To understand how DocPrism performs in detecting incorrectness inconsistencies across multiple programming languages, we

built a dataset including 20 projects, aiming for 100 documented functions per project. 5 projects were written in each of four programming languages: Python, TypeScript, C++, and Java. We selected 5 popular open-source projects[4] for each language by the following protocol:

(1) Sorting by the most starred projects on GitHub.
(2) The project must have at least two years of development history.
(3) The project must be under active development, with at least 5 issues or pull requests closed in the last 2 months.
(4) At least 60% of the project's source code is written in the target programming language, according to the GitHub repository statistics.

From each of the selected 20 projects, we randomly sampled 100 functions that satisfy the following requirements:

- The function has a method-level documentation that follows documentation syntax for the given language (e.g., Javadoc).
- The function is implemented in the language of interest.
- Both the function and its documentation have more than 7 tokens.
- There are no duplicate code-documentation pairs.

Unfortunately, two projects had fewer than 100 methods meeting these criteria. Only 94 functions from the TypeScript-based Nest project and 97 functions from the C++-based llama.cpp project met these criteria. The final dataset contains 1,991 code-documentation pairs: 500 in Python, 494 in TypeScript, 497 in C++, and 500 in Java. The **implication** of this dataset is that it provides an authentic view into code-documentation pairs as they exist in deployed software systems. This allows us to examine these pairs for real-world inconsistencies. However, we cannot automatically assign consistent/inconsistent labels to the code-documentation pairs. Since they are in popular production software, we conjecture that *many* of them are consistent, but we cannot know a priori which ones are.

To address this challenge, we invested substantial effort to manually label a subset of this data. Determining whether a function and its documentation are consistent is time-consuming work. To make this process tractable, we labelled only those code-documentation pairs that DocPrism flagged as inconsistent. This was a more direct process as the tool provides rationale and explicitly-highlighted evidence from the documentation and code that a labeller can assess as either correct or incorrect. Labelling a DocPrism-flagged code-documentation pair takes between 5-15 minutes: longer times are usually because we need to analyze code in the repository outside the core function (i.e., its callees) to determine whether the inconsistency is correct. Each code-documentation pair was labelled by a single author. The **implication** of this labelling methodology was that we are able to assess DocPrism's precision (as all flagged pairs are labelled for true/false positive analysis), but cannot assess recall (as false negatives were not examined). Since only a single labeller examined each output, consistency could be a concern.

To mitigate the single-coder labelling threat, we performed an inter-coder reliability analysis. This evaluates how consistently different labellers would assess the same code-documentation pairs. Two authors independently labelled 200 randomly chosen flagged functions, 50 from each of four programming languages: Python, TypeScript, Java, and C++. The Cohen's Kappa for these labels is 0.737, which is classified as *substantial* agreement [11]. The **implication** of this is that individual labellers can reliably determine whether DocPrism-flagged code-documentation pairs in our dataset are inconsistent.

[4]Details of the 20 projects are included in the linked artifact.

5.2.2 Ablation dataset (Python). Our ablation dataset contains 52 fully labelled code-documentation pairs in Python. These pairs were randomly selected from two popular Python projects.[5] The 52 functions were chosen using the same sampling methodology described in the primary dataset. Unlike the primary dataset, we labelled all 52 functions *a priori*: DocPrism did not classify them first. Additionally, all 52 functions were independently labelled by two authors. The code-documentation pairs are labelled as either consistent or inconsistent; when inconsistent, the source of the inconsistency is noted. Any differences in labels were discussed among both coders until agreement was reached for all 52 code-documentation pairs. The **implication** of this dataset is that this fully-labelled dataset allows us to investigate DocPrism in greater detail by measuring precision, recall, accuracy, and F1 score across ablation variants.

5.3 Evaluation Metrics

Our metrics have two levels of granularity. *Function-level* metrics evaluate DocPrism's on a per-function basis. *Inconsistency-level* metrics evaluate each identified inconsistency separately.

5.3.1 Function-Level Terminology.

(1) **True Positive (TP):** The function is inconsistent with its documentation, and DocPrism surfaces at least one of the inconsistencies.
(2) **False Positive (FP):** The function is consistent with its documentation, but DocPrism surfaces at least one inconsistency.
(3) **True Negative (TN):** The function is consistent with its documentation, and DocPrism surfaces no inconsistencies.
(4) **False Negative (FN):** The function is inconsistent with its documentation, but DocPrism surfaces no inconsistencies.
(5) **Flag Rate:** The percentage of functions flagged as inconsistent with their method-level docs.

$$\text{Flag Rate} = \frac{\#\text{ Functions Flagged as Inconsistent}}{\#\text{ Total Tested Functions}}$$

5.3.2 Inconsistency-Level Terminology.

(1) **True Positive (TP):** The reported inconsistency is indeed an inconsistency between code and documentation.
(2) **False Positive (FP):** The reported inconsistency is wrong.
(3) **Under-promise Rate (U.P. Rate):** The percentage of *under-promises* in the final output.

$$\text{Under-Promise Rate} = \frac{\#\text{ Under-Promises}}{\#\text{ Total Reported Inconsistencies}}$$

5.3.3 C4RLLaMA Function-Level Terminology. When C4RLLaMA [17] flags a code-doc pair as inconsistent, it does not explain the inconsistency, but directly outputs rectified documentation. Labelling C4RLLaMA on whether its rectification is correct would be more strict than our true positive above. To align our measurement as much as possible with our function-level metrics, we use the following definitions of function-level true- and false-positive:

(1) **True Positive (TP):** C4RLLaMA identifies the function as inconsistent with its documentation, and updates at least one documentation fragment where an inconsistency occurs. Note it need *not* correctly rectify the inconsistency.
(2) **False Positive (FP):** C4RLLaMA identifies the function as inconsistent with its documentation, but does not update any documentation fragment in which there is an inconsistency.

For both DocPrism and C4RLLaMA, we consider only *incorrectness* inconsistencies in the definitions.

[5]Pandas: https://github.com/pandas-dev/pandas; Requests: https://github.com/psf/requests

6 Results

We aim to answer the following questions:

RQ1. How does DocPrism compare with the state-of-the-art C4RLLaMA?
RQ2. How does DocPrism perform on real-world code-doc inconsistencies across multiple programming languages? What are its sources of false positives?
RQ3. How prevalent are code-doc incorrectness inconsistencies in practice? What kinds of incorrectness inconsistency does DocPrism find?
RQ4. How do our design decisions, in particular *Local Categorization and Exernal Filtering* (LCEF), affect DocPrism's performance?
RQ5. How does LCEF generalize to other LLMs?

6.1 RQ1: DocPrism vs. State-of-the-Art

As mentioned in Section 2.1, C4RLLaMA [17] is the state-of-the-art (SOTA) in both *just-in-time* and *post-hoc* inconsistency detection on the Panthaplackel dataset [13], in Java. All of the dataset's inconsistent examples are code-comment pairs that never appear together in the real codebase (ref. Section 2.4), i.e., they are synthetic. The setting we target in this work is the *post-hoc* setting. We used the scripts provided in the C4RLLaMA replication package to fine-tune CodeLLaMA. We use the evaluation setup described in Section 5.1. Detailed outputs can be found in our artifact.

6.1.1 DocPrism Dataset: Flag Rates. We compare DocPrism with C4RLLaMA on our Java dataset, which contains 500 code-documentation pairs as they appear in real-world codebases (ref. Section 5.2) We refer to it as the DocPrism dataset. We summarize DocPrism and C4RLLaMA's flag rates in Table 1. The flag rate represents the proportion of functions flagged by a tool as inconsistent with its documentation. DocPrism takes code and the entire documentation block as input, which may contain multiple components (e.g., Summary, @param, @return). DocPrism flags 92 of 500 real-world code-documentation pairs as inconsistent, an 18% flag rate.

The dataset on which C4RLLaMA was trained and evaluated, however, separates this full documentation block into individual documentation components (i.e., Summary, @param, @return). To make our dataset more like this, we extracted the Summary, @param, and @return components (if they exist) from the full documentation, and pair the corresponding source code with each of them. This led to a total of 1,159 pairs: 464 ⟨code, @param only⟩, 234 ⟨code, @return only⟩, and 461 ⟨code, summary only⟩. Then, we ran C4RLLaMA on these three types of pairs.

With the documentation split in this way, we see C4RLLaMA flag something as consistent. C4RLLaMA flags 463 out of 464 ⟨code, @param only⟩, all 234 ⟨code, @return only⟩ and all 461 ⟨code, summary only⟩ pairs as inconsistent. But this still leads to a 100% flag rate on code-documentation pairs as a whole—one of the documentation parts was flagged for each code-documentation pair.

Table 1. Flag rate comparison between DocPrism and state-of-the-art across four types of pairs on the Java dataset (Section 5.2). ⟨code, *X* only⟩ corresponds to the flag rate for input pairs of code and individual documentation component *X*. *Full documentation* includes all available documentation components. *%Flagged code-doc pairs* indicates the percentage of code-documentation pairs flagged as inconsistent.

	DocPrism	**C4RLLaMA**
⟨code, full documentation⟩	18.4% (92/500)	100 % (500/500)
⟨code, @param only⟩	–	99.8% (463/464)
⟨code, @return only⟩	–	100 % (234/234)
⟨code, summary only⟩	–	100 % (461/461)
% Flagged code-doc pairs	18.4% (92/500)	100 % (500/500)
# Flagged inconsistencies	175	500 (full doc) or 1,158 (components)

6.1.2 DocPrism Dataset: Manual Precision. We further examine C4RLLaMA's precision on this dataset. The 100% flag rate means we cannot apply our manual labelling methodology on *all* of the 1,158 inconsistencies flagged by C4RLLaMA. Instead, we randomly sampled 50 code-documentation pairs from the DocPrism dataset that either DocPrism or C4RLLaMA flagged as inconsistent. This gives us a total of 50 ⟨code, documentation⟩, 47 ⟨code, Summary⟩ , 32 ⟨code, @param⟩, and 24 ⟨code, @return⟩ pairs. We chose a sample size of 50 because labelling C4RLLaMA outputs is more time-consuming than labelling DocPrism outputs. To reduce sample size threats, we report precision results with a 95% Wilson score confidence interval [4].

We label DocPrism and C4RLLaMA outputs using the true- and false-positive definitions introduced in Section 5.3.1 and Section 5.3.3 respectively. Two authors independently label the C4RLLaMA outputs, and the remaining authors discuss and resolve any labelling disagreements.

As shown in Table 2, under our measure of precision, C4RLLaMA achieves a precision of 0.08 across all pairs, with a 95% Wilson score confidence interval of 0.05 – 0.14. In contrast, DocPrism achieves a precision of 0.58 [0.47 – 0.67], based on our manual labels for all flagged pairs (ref. RQ2 in Section 6.2). A precision of 0.05 – 0.14 is much lower than the 0.94 reported by C4RLLaMA on the Pathaplackel dataset [17]. Next, we investigate performance on the Pathaplackel data to determine whether the change in precision is due to the dataset shift or the different measure of precision.

Table 2. Precision of DocPrism and C4RLLaMA on the Panthaplackel and DocPrism datasets under our precision measures. Ranges indicate 95% confidence intervals [4].

	Panthaplackel Dataset	**DocPrism Dataset**
DocPrism	0.78 [0.62 - 0.88]	0.58 [0.47 - 0.67]
C4RLLaMA	0.83 [0.67 - 0.92]	0.08 [0.05 - 0.14]

6.1.3 Panthaplackel Dataset and Metrics. We run C4RLLaMA and DocPrism on the *Panthaplackel [13] dataset*. This dataset provides consistent/inconsistent labels: Table 3 presents the results in terms of these labels. We successfully reproduce the C4RLLaMA results reported in their paper [17], averaging a 2.7%[6] difference across the metrics, using their scripts and the fine-tuned model. Overall, C4RLLaMA achieves higher metric values than DocPrism (e.g., 0.87 vs. 0.67 in precision). However, this does not necessarily indicate that it outperforms DocPrism on the Panthaplackel dataset. The reasons are twofold. First, human analysis on a subset suggests that 17-20% of the labels in this dataset are incorrect [13]. Second, as the labels are binary, the metric does not measure whether the surfaced inconsistencies are *correct*, just that *an* inconsistency is surfaced.

6.1.4 Panthaplackel Dataset: Manual Precision. To answer whether C4RLLaMA's 0.08 precision on our dataset is due to the dataset change or our precision measure, we apply our precision measure to the Panthaplackel Dataset. The Panthaplackel dataset does not contain full ⟨code, documentation⟩ pairs, as DocPrism expects. We consider ⟨code, Summary only⟩ pairs, as these are closest to full documentation. Thus, we perform our labelling on 50 randomly-sampled ⟨code, Summary only⟩ pairs that either DocPrism or C4RLLaMA flagged as inconsistent. Again, two authors independently label the C4RLLaMA and DocPrism outputs using the true- and false- positive definitions from Sections 5.3.1 and 5.3.3. The remaining authors discuss and resolve the labelling disagreements.

The left column of Table 2 shows the results. Under our measure of precision, C4RLLaMA achieves a precision of 0.83 on the Panthaplackel dataset, (95% confidence interval [4] : 0.67 – 0.92). This is comparable to C4RLLaMA's precision on the same dataset with its own measure of precision: 0.89. From this, we conclude that, C4RLLaMA's precision drop to 0.08 on the DocPrism dataset is due to the *dataset change* rather than *our precision measure*. In Table 2, DocPrism shows more stable performance than C4RLLaMA across the two datasets. While DocPrism performs comparably to

[6]Results vary across runs, with differences ranging from 1.3% to 2.7%.

Table 3. Performance of DocPrism and C4RLLaMA on the Panthaplackel dataset using its binary labels.

	DocPrism					C4RLLaMA				
	Flag rate	**Precision**	**Recall**	**Accuracy**	**F1**	**Flag rate**	**Precision**	**Recall**	**Accuracy**	**F1**
⟨code, @param only⟩	52%	0.62	0.64	0.63	0.63	47%	0.92	0.87	0.89	0.89
⟨code, @return only⟩	43%	0.7	0.6	0.67	0.65	48%	0.9	0.87	0.89	0.89
⟨code, summary only⟩	38%	0.68	0.51	0.63	0.58	51%	0.85	0.86	0.85	0.85
Total	44%	0.67	0.59	0.65	0.63	49%	0.89	0.87	0.88	0.88

Table 4. DocPrism performance across four programming languages: Python, TypeScript, C++, and Java; see Section 5.3 for explanations of all metrics. *Flag rate* is the percentage of functions flagged as inconsistent. *#Inconsistencies* is the total number of inconsistencies reported by DocPrism. *U.P. rate* is *under-promise rate.*

	Function-level data					Inconsistency-level data				
	# Functions	**Flag rate**	**# TP**	**# FP**	**Precision**	**# Inconsistencies**	**U.P. rate**	**# TP**	**# FP**	**Precision**
Python	500	19%	61	32	0.66	176	2%	105	71	0.60
TypeScript	494	16%	57	24	0.70	134	5%	89	45	0.66
C++	497	14%	40	31	0.56	123	8%	67	56	0.55
Java	500	18%	53	39	0.58	175	3%	91	84	0.52
Total†/ Mean	1,991†	17%	211†	126†	0.63	608†	4%	352†	256†	0.58

C4RLLaMA on the Panthaplackel dataset (DocPrism: 0.78, C4RLLaMA: 0.83), it achieves much higher precision on the DocPrism dataset (DocPrism: 0.58, C4RLLaMA: 0.08).

At a higher level, good performance on the Panthaplackel dataset, under both measures of precision, does not translate to higher performance on the DocPrism dataset. We recommend future work evaluating on synthetically created code-doc pairs also report results code-doc pairs as they exist in real-world codebases.

RQ1 Takeaway. Using our manual precision metrics, DocPrism performs comparably to C4RLLaMA (precision: 0.78 vs 0.83, flag rate: 44% vs 49%) on the Panthaplackel dataset. On our DocPrism dataset, the tools perform very differently (precision: 0.47-0.67 vs 0.05-0.14, flag rate: 18% vs 100%). C4RLLaMA's shift in performance suggesst there is a distribution shift between synthetic cross-commit inconsistencies and naturally-occuring ones.

6.2 RQ2: Inconsistency Detection Effectiveness

In RQ1 we examined only inconsistency flag rate. The natural follow-up question is: are the flagged issues true inconsistencies? To answer this, we run DocPrism on 1,991 developer-written code-documentation pairs over four programming languages: Python, TypeScript, C++, and Java (§ 5.2). We manually analyze DocPrism's detected inconsistencies to determine whether they are correct. Further, we categorize DocPrism's most common sources of false positives.

6.2.1 Results. We manually inspect each inconsistency DocPrism reports to determine whether it is a true positive (a correctly identified inconsistency) or a false positive (an invalid inconsistency).[7] From this, we can calculate precision (§ 5.3).

Table 4 shows the results. We split results into function-level and inconsistency-level. This is because DocPrism can return multiple inconsistencies for a single function (see § 5.3 for details).

Overall, DocPrism performs well on *flag rate* and *under-promise rate* across all four languages. The *under-promise* rate ranges from 2% to 8%, showing that DocPrism mostly surfaces the targeted *incorrectness* inconsistencies. The flag rate ranges between 14% to 19%. We conjecture this is more

[7]This manual analysis is detailed in Section 5.

tractable for human analysis than the 100% flag rate of state-of-the-art (§ 6.1). DocPrism achieves higher precision on TypeScript and Python (0.7, 0.66), than on C++ and Java (0.56, 0.58).

6.2.2 Sources of DocPrism False Positives. Table 5 summarizes the four main reasons for DocPrism's false positives across the examined languages.

Lack of API knowledge (35%): The LLM is unsure about, or assumes missing, functionality in a callee API, even though the callee API actually implements the desired behaviour. The `info()` example from Fig. 5 falls in this category. The documentation states that the bug information should be pretty-printed as JSON. However, the function calls `info()` to get the bug information. As shown in the explanation box, the LLM cannot determine whether `info()` generates bug information, and thus flags this as inconsistent.

```
def main():
    """Pretty-print the bug information as JSON."""
    print(json.dumps( info() , sort_keys=True, indent=2))
```

Explanation generated by DocPrism: *The code does not explicitly handle 'bug information'. The info() function is called, but its implementation and relation to bug information are unclear.*

Fig. 5. An example of a false positive reported by DocPrism due to *lack of API knowledge* (`Requests-26`).

Semantics Gap (7%): The LLM correctly characterizes the semantics of the documentation and code snippet, but fails to identify that the implementation in the code is an instance of the semantics in the documentation, or vice-versa. For example, the documentation of `n8n-43` mentions a feature of "expiring the current client" and the code implements it with `this.client = undefined;`. The LLM flags this function as inconsistent, saying that *"the documentation mentions expiring the current client, but the code only sets it to undefined"*. The LLM fails to identify that setting to `undefined` is sufficient to meet the definition of "expire".

Under-Promise (10%): The LLM points out that some details are not documented (§ 3.3).

Incorrect Reasoning (48%): The LLM incorrectly characterizes code or documentation semantics, or provides erroneous explanations. This is distinct from *Semantics Gap*, where the LLM correctly characterizes the semantics of both the code and the documentation, but fails to align them. We divide this into four subcategories: *temporal constraints*, *contextual info*, *no domain knowledge*, and *others*. *Temporal constraints* refer to cases where the documentation sets additional requirements regarding the usage of this function, but the LLM misinterprets them as features that need to be implemented in the code. The same applies to *contextual info*. For example, the documentation of `Tesseract-48` mentions some surrounding context for the function: *"this routine is designed to be used in concert with the KDWalk routine."* As this synchronization is not explicitly implemented in the function body, the LLM flags this as an inconsistency. *No domain knowledge* means that detected inconsistency is wrong due to a lack of relevant domain-specific knowledge.

RQ2 Takeaway. DocPrism almost always surfaces incorrectness inconsistencies (92%-97%). It achieves higher precision on TypeScript and Python (0.7, 0.66), than on C++ and Java (0.56, 0.58). Most false positives are due to *incorrect reasoning*, then *lack of API knowledge*.

6.3 RQ3: Prevalence of Incorrectness Inconsistencies

In the process of manually labelling all DocPrism-flagged inconsistencies for RQ2, we found that incorrectness inconsistencies do exist in maintained software systems. Of the 1,991 code-doc pairs on which we ran DocPrism, we confirmed that there were incorrectness inconsistencies for 211 of them (Table 4, *# TP*). This conservatively suggests that there exist errors in the code or documentation of at least 11% of non-trivial documented methods.

Table 5. Manual categorization of *incorrectly* identified inconsistencies surfaced by DocPrism

	Python	TS	C++	Java
Lack of API Knowledge	39 (55%)	15 (33%)	17 (30%)	18 (21%)
Semantics Gap	1 (1%)	4 (9%)	2 (4%)	10 (12%)
Under-promise	4 (6%)	7 (16%)	10 (18%)	5 (6%)
Incorrect Reasoning	27 (38%)	19 (42%)	27 (48%)	51 (61%)
Domain Knowledge	2	3	0	0
Temporal Constraints	4	0	1	6
Contextual Info	5	5	1	18
Others	16	11	25	27
Total	71	45	56	84

Table 6. Manual categorization of inconsistency errors *correctly* surfaced by DocPrism.

	Python	TS	C++	Java
Functionality Mismatch	58 (55%)	42 (47%)	38 (57%)	29 (32%)
Parameter Mismatch	7 (7%)	5 (6%)	2 (3%)	11 (12%)
Parameter Type Mismatch	1 (1%)	7 (8%)	0 (0%)	4 (4%)
Identifier Mismatch	0 (0%)	9 (10%)	3 (5%)	2 (2%)
Return Type Mismatch	3 (3%)	11 (12%)	4 (6%)	1 (1%)
Unimplemented Feature	35 (33%)	15 (17%)	10 (15%)	44 (48%)
Unused Parameter/Variable	1 (1%)	0 (0%)	10 (15%)	0 (0%)
Total	105	89	67	91

Table 6 summarizes the seven main sources of code-doc inconsistencies we identified. They are:

Functionality Mismatch (47%): Features described in the documentation logically conflict with their implementation. Fig. 2 gives an example: while the documentation mentions returning the highest `OrderIndex + 1`, the code implements returning the highest `OrderIndex`.

Parameter Mismatch (7%): Either some documented parameters are not present in the function signature, or only a subset of the parameters defined in the function signature are documented.

Parameter Type Mismatch (3%): Param. types in the docs do not match those in the signature.

Identifier Mismatch (4%): Different identifiers refer to the same object in the code vs. the docs.

Return Type Mismatch (5%): The function's return type differs from what is in the docs.

Unimplemented Feature (30%): A feature specified in the docs is absent from the code.

Unused Parameter or Variable (3%): Unlike *parameter mismatch*, the parameters specified in the documentation do match with those defined in the function signature. However, some parameters are not used in the code body at all; or, some C++ codebases use `GGML_UNUSED(variable)` to avoid compiler warnings.

The most prominent source of error varies by language. was *Unimplemented Feature* the most prevalent in Java, *Functionality Mistmatch* was most common for Python, TypeScript and C++.

RQ3 Takeaway. 11% of developer-written code-documentation pairs in the multi-language dataset exhibit incorrectness inconsistencies. This provides a conservative lower-bound on the prevalence of developer-written code-documentation inconsistencies in real-world projects.

6.4 RQ4: Ablation Studies

In this section, we evaluate the impact of different prompt variants on DocPrism's performance. Table 7 shows the six variants of DocPrism; they use different sets of building blocks, each defined below. The goal is to understand the impacts of two main design decisions in DocPrism: *local categorization* (LC) and *external filtering* (EF). To understand LC, we look at the impact of chain-of-thought, as LC is akin to a specific form of chain-of-thought. To understand EF, we contrast with a more typical prompt engineering strategy of instructed filtering.

Table 7. Prompt and technique components in each studied variant. The DC variant is DocPrism.

Variant	Prefix	Plain JSON	CoT JSON	Ins. Cat.	Local Cat.	Ins. Filter	Ext. Filter	Type Label	Summary
V1	✓	✓							
V2	✓		✓						
V3	✓	✓		✓		✓			
V4	✓		✓	✓		✓			
V6	✓		✓	✓			✓	✓	
V7	✓		✓	✓			✓	✓	✓
DC	✓		✓		✓		✓		✓

6.4.1 Variant Building Blocks. We consider nine building blocks for variant construction.

(1) **Prefix:** As explained in Section 4.1.1, *Prefix* sets the model character, specifies the task, and indicates the input format. It is prefixed in all prompt variants.
(2) **Plain JSON:** Unlike *CoT JSON*, *Plain JSON* does not require the LLM to perform any chain-of-thought reasoning while generating outputs. It just needs to fill in the following key.

```
{"identified_inconsistency": "..."}
```

(3) **CoT JSON:** For each inconsistency, the LLM must do chain-of-thought [24] by: identifying which documentation/code snippets contain conflicting information, and provide an explanation.

```
{"original_documentation_snippet_that_has_conflicting information_with_code": "...",
"original_code_snippet_that_has_conflicting_information_with_documentation": "...",  "explanation": "..."}
```

(4) **Instructed Categorization:** The prompt defines our three categories of inconsistencies (§ 3).

```
You should focus on identifying three types of inconsistencies.
1. Over-promise: Some core parts of the documentation are not implemented in the code.
2. Direct mismatch: The code does not correctly implement what is mentioned in the documentation.
3. Under-promise: Some code is not documented or mentioned in the documentation.
```

(5) **Local Categorization:** Our JSON-based Local Categorization approach (Subtasks 3-5 in § 4.1.2).
(6) **Instructed Filter:** The prompt explicitly instructs the LLM not to report *under-promises*.

```
Do not report inconsistencies in the third category, under-promise, in the final output.
```

(7) **External Filter:** An external filtering module takes in the LLM output and filters out *under-promises* during the post-processing stage.
(8) **Type Label:** The LLM is must explicitly label the type of each inconsistency as either *over-promise*, *direct mismatch*, or *under-promise*.

```
{"inconsistency_type": "...", <other keys required for CoT JSON>}
```

(9) **Code/Doc Summary:** The LLM is required to fill in two keys, `Documentation_Summary` and `Code_Summary`, before identifying and listing inconsistencies.

Table 8. DocPrism vs. ablations on ablation dataset. *F. Precision* refers to *Function-level Precision*. *I. Precision* denotes *Inconsistency-level Precision*. *U.P. rate* means *under-promise rate*; the lower, the better. Definitions of evaluation metrics can be found in Section 5.3.

	V1	V2	V3	V4	V6	V7	DocPrism
Flag rate	94%	94%	96%	98%	48%	39%	14%
F. Precision	0.12	0.12	0.1	0.12	0.2	0.1	0.71
Recall	1	1	1	1	0.83	0.33	0.71
Accuracy	0.17	0.17	0.14	0.14	0.6	0.58	0.94
F1 score	0.22	0.22	0.18	0.21	0.32	0.15	0.77
U.P. rate	72%	44%	48%	28%	27%	50%	0%
I. Precision	0.07	0.14	0.07	0.1	0.17	0.09	0.7

6.4.2 Results. Table 8 presents the ablation study results on the ablation dataset across seven evaluation metrics. Table 10 shows the flag rates on the Python, TypeScript, C++, and Java datasets. Table 9 and Table 12 report McNemar's test [12] results and Cohen's g [2] values to evaluate DocPrism's improvements over the ablation variants in terms of accuracy and flag rate. All p-values are well below 0.01. Cohen's g values range from 0.28 to 0.50, exceeding the conventional threshold of 0.25 for a large effect. This indicates that DocPrism's improvements over the ablation variants in flag rate and accuracy are both statistically significant and practically substantial.

Table 9. Statistical evaluation of DocPrism's improvements over ablation variants using McNemar's test and Cohen's g, based on the results in Table 8. Cohen's g ranges from 0 to 0.5; values of 0.25 or higher represent *large* effect sizes.

	DocPrism vs. V4		DocPrism vs. V7	
	p-value	**Cohen's g**	**p-value**	**Cohen's g**
Flag rate	1.1×10^{-13}	0.5	1.1×10^{-2}	0.28
Accuracy	1.1×10^{-10}	0.45	6.6×10^{-5}	0.4

Table 10. DocPrism vs. ablations: flag rates on the main Python, TypeScript, C++, and Java datasets across ablation variants.

	V1	V2	V3	V4	V6	V7	DocPrism
Python	95%	95%	97%	97%	95%	56%	19%
TypeScript	93%	90%	93%	94%	58%	52%	16%
C++	90%	90%	93%	92%	62%	60%	14%
Java	94%	92%	91%	91%	67%	57%	18%

Table 11. DocPrism vs. ablations: flag rates across ablation variants with GPT4.1. ($\pm X\%$) refers to the difference between the adjacent value and the corresponding one in Table 10.

	V4	V7	DocPrism
Python	91%(-6%)	60%(+4%)	24%(+5%)
TypeScript	82% (-12%)	47% (-5%)	17% (+1%)
C++	90% (-2%)	51% (-9%)	20% (+6%)
Java	86% (-5%)	60% (+3%)	24% (+6%)

Table 12. Statistical evaluation of DocPrism's improvements over ablation variants using McNemar's test and Cohen's g, based on the results in Table 10. Cohen's g ranges from 0 to 0.5; values of 0.25 or higher are typically considered large effects.

	DocPrism vs V4		DocPrism vs V7	
	p-value	**Cohen's g**	**p-value**	**Cohen's g**
Python	2.0×10^{-118}	0.5	1.2×10^{-43}	0.4
TypeScript	4.9×10^{-114}	0.5	9.7×10^{-43}	0.4
C++	2.5×10^{-116}	0.5	5.1×10^{-54}	0.4
Java	2.4×10^{-108}	0.5	3.2×10^{-43}	0.4

Table 13. Statistical evaluation of DocPrism's improvements over ablation variants using McNemar's test and Cohen's g, based on the results in Table 11. Cohen's g ranges from 0 to 0.5; values of 0.25 or higher are typically considered large effects.

	DocPrism vs V4		DocPrism vs V7	
	p-value	**Cohen's g**	**p-value**	**Cohen's g**
Python	5.7×10^{-101}	0.5	2.1×10^{-47}	0.47
TypeScript	7.5×10^{-96}	0.5	8.4×10^{-43}	0.49
C++	8.5×10^{-109}	0.5	2.1×10^{-50}	0.5
Java	1.5×10^{-92}	0.5	4.2×10^{-50}	0.49

Compared to all six variants, DocPrism achieves up to a 5.1× improvement in F1 score, 6.7× in accuracy, 7.1× in function-level precision, and 10× in inconsistency-level precision on the ablation dataset. It also reduces the *under-promise* rate from 72% (V1) to 0% and the flag rate from 98% (V4) to 14%. Similarly, the average flag rate on Python, TypeScript, C++, and Java goes from 94% to 17%.

Effects of Chain-of-Thought (V1 vs. V2 / V3 vs. V4). *CoT JSON* guides the LLM use chain-of-thought reasoning while generating outputs. V2 adds *CoT JSON* to V1, and V4 adds *CoT JSON* to V3. Adding *CoT JSON* reduces under-promise rates in both cases: from 72% (V1) to 44% (V2), and from 48% (V3) to 28% (V4). We also see consistent improvements in inconsistency-level precision, going up from 0.07 (V1) to 0.14 (V2), and from 0.07 (V3) to 0.10 (V4).

Effects of Instructed Categorization and Instructed Filter (V1 vs. V3 / V2 vs. V4). V3 and V4 add *Instructed Categorization* and *Instructed Filter* to V1 and V2, respectively. This enhances the prompt with the three inconsistency categories and instructions to filter out *under-promises*. We do see the under-promise rate reduce: from 72% (V1) to 48% (V3) and from 44% (V2) to 28% (V4). However, the LLM still struggles to mitigate the under-promise rate: 48% and 28% is still quite high.

Effects of External Filter (V6 vs. V4). Instead of telling the LLM not to report *under-promises* through the prompt (V4), V6 prompts the LLM to label each detected inconsistency with its category (i.e., *over-promise*, *direct mismatch*, or *under-promise*). It then filters out inconsistencies labelled with *under-promise* during the post-processing stage. V6 reduces the average flag rate from 92% to 62% on the TypeScript, C++, and Java datasets. It improves six out of seven evaluation metrics

on the ablation dataset. Notably: reducing the flag rate from 98% (V4) to 48% (V6) and increasing accuracy from 0.14 to 0.6—due to a substantial increase in true negatives. But, there is no significant decrease in the under-promise rate. This is partly because V6 mislabels some *under-promises* as *direct mismatches*, so that some real *under-promises* cannot be filtered out. In contrast, DocPrism, with the *Local Categorization*, substantially reduces the instances of mislabelling.

Effects of Local Categorization (DocPrism vs. V7) V7 adds documentation and function summaries to V6. Like V6 and V7, DocPrism employs an *external filter* to drop under-promises during the post-processing stage. However, unlike V6/V7, DocPrism adopts *local categorization* (LC), where the LLM is not required to "remember" the definitions of the inconsistency categories. Instead, it must answer the categorization questions for each check-in key in the predefined JSON schema. DocPrism outperforms V7 on all evaluation metrics, and V6 on all metrics except recall. It reduces the flag rate from 39% to 14% and the under-promise rate from 50% to 0%, and increases function-level precision from 0.1 to 0.71, accuracy from 0.58 to 0.94, F1 score from 0.15 to 0.77, and inconsistency-level precision from 0.09 to 0.70. This shows the effectiveness of *local categorization*.

RQ4 Takeaway. *Local Categorization, External Filtering* (LCEF) is highly effective at reducing DocPrism's under-promise and flag rates. Both LC and EF greatly decrease flag rates, but LC is key to decreasing under-promise rate and increasing precision.

6.5 RQ5: Generalization to Other LLMs

The previous research question (ref. Section 6.4) shows that the *Local Categorization, External Filtering* (LCEF) approach using LLaMA3.1-70B is highly effective at reducing flag rates. This research question investigates how our approach generalizes to other LLMs.

We select two representative ablation variants introduced in Section 6.4: V4 and V7. V4 instructs LLMs to filter out *under-promise* inconsistencies in the prompt (i.e., *Instructed Filter*). V7 replaces the *Instructed Filter* with the *Type Label* and *External Filter* components, i.e., asks the LLM to explicitly categorize each inconsistency. We run these variants and DocPrism using GPT4.1 on our datasets.

Table 11 shows the results. We see the same trend of decreased flag rate, from V4 to V7 to DocPrism, observed with LLaMA3.1-70B in RQ4 (ref. Section 6.4). We use McNemar's test [12] and Cohen's g [2] to evaluate DocPrism's flag-rate improvements over the ablation variants. As shown in Table 13, all p-values are near zero, and Cohen's g ranges from 0.47 to 0.50, exceeding Cohen's g threshold for large effect sizes (0.25). Thus, the flag-rate improvements achieved by DocPrism-GPT4.1 are both statistically significant and practically substantial.

Comparing Table 10 with Table 11, we see that DocPrism-GPT4.1 has slightly higher flag rates than DocPrism-LLaMA3.1. But V4-GPT4.1 has lower flag rates than V4-LLaMA3.1. This suggests the standard "define categories and instruct to filter" approach is more effective on GPT4.1. Overall, the results show that a newer-generation model still suffers from the high flag rate we observed with LLaMA3.1, and that LCEF is still effective at reducing it, regardless of the model used.

RQ5 Takeaway. Even a proprietary model (GPT4.1) has unacceptably high flag rates when using standard prompting techniques. DocPrism is effective at reducing high flag rates with this proprietary model, suggesting it is not over-fit to one specific model.

7 Discussion

Our evaluation shows that code-documentation inconsistency errors are not uncommon in real-world developer-written code (RQ3). Our ablation study (RQ4) demonstrates how LCEF dramatically decreases flag- and under-promise rates, but also greatly improves accuracy to over 90%. We think

low flag and under-promise rates are important for developers wanting to use a tool detecting code-documentation inconsistencies—otherwise they will be overwhelmed with unimportant warnings.

DocPrism performance contrasts starkly with our evaluation of SOTA (RQ1). C4RLLaMA achieved 94% precision and 90% accuracy on a widely-used, but synthetic, dataset. But, it flagged every function in our dataset—made of code-documentation pairs that co-occur in real codebases—as inconsistent with its documentation, leading to a precision of only 0.05-0.14. This result aligns with our ablation study, where our first uses of LLMs to find inconsistencies had flag rates over 90%. However, it shows a danger with relying on automatically-labelled datasets and binary labels in building software engineering tools.

Ranking results. Given the surprisingly large numbers of inconsistencies found in real-world code (RQ3), flagging this many results for a developer could still be overwhelming. We could reduce this by ranking inconsistencies, to help developers focus on only those that are most impactful. This would help them to maximize any effort invested into improving the documentation consistency within their project. Such improvements could help both future developers reading the documentation, as well as any emerging LLM-based development tools that rely on accurate documentation as the basis for constructing prompts for generating implementations.

Overlap between different inconsistencies. When examining our results, we noted that DocPrism sometimes outputs the same inconsistency into two categories. This happens most often with an inconsistency being listed as both an over-promise and direct mismatch, though sometimes an under-promise was also listed as direct mismatch (e.g., Fig 3). This suggests that Local Categorization's effectiveness depends on how well the Check-In Key question describes the target category. Upon analysis, our Check-In Key for direct mismatches, *"Does the code correctly implement what is mentioned in the documentation?"*, is semantically vague: "correctly implement" can also capture the notion of over-promise. This suggests that the success of LCEF may depend on whether a brief, unambiguous check-in key can be constructed.

8 Conclusion

In this paper, we have introduced DocPrism, which flags and explains incorrectness inconsistencies between functions and their documentation. Examining developer-written source code and their documentation from 20 real-world repositories across four programming languages, DocPrism flags a low number of inconsistencies (< 18%), meaning developers do not need to examine many results from this analysis. This is much lower than the over 90% flag rates of standard prompting techniques, or 100% flag rates of state-of-the-art. Our approach demonstrates that detecting inconsistency errors between real-world developer-written function-level documentation and their source code is possible. We believe tools such as DocPrism will enable developers to better trust the documentation in their systems and once again rely on documentation as an accurate and useful form of abstraction crucial for building and evolving their systems.

Acknowledgments

Thanks to the anonymous reviewers whose suggestions have improved the paper. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [CRC-2024-00299, RGPIN-2023-04478].

9 Data Availability

The artifact is available at https://github.com/SnowPhoebe/DocPrism and https://doi.org/10.5281/zenodo.21449177. It includes the datasets containing all of the functions and documentation we examined, the prompts (including ablation prompts), DocPrism outputs for the tested functions, baseline outputs, our manually-determined labels, and details of the 22 projects from which the

datasets were collected. It also contains representative examples for each true positive or false positive category, as identified by our tool DocPrism. Examples are identified as codebase-n, corresponding to our n^{th} function in that codebase's dataset.

References

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1199–1210. doi:10.1109/ICSE.2019.00122

[2] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge.

[3] Zhanqi Cui, Shifan Liu, Li Li, and Liwei Zheng. 2025. SEOCD: Detecting obsolete code comments by fusing semantic features and expert features. *Expert Systems with Applications (ESWA)* 280 (2025), 127470. doi:10.1016/j.eswa.2025.127470

[4] Kevin Dunnigan. 2008. Confidence interval calculation for binomial proportions. In *Proceedings of the SAS Global Forum (MWSUG)*.

[5] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024). doi:10.48550/arXiv.2407.21783

[6] Yuan Huang, Yinan Chen, Xiangping Chen, and Xiaocong Zhou. 2025. Are your comments outdated? Toward automatically detecting code-comment consistency. *Journal of Software: Evolution and Process* 37, 1 (2025), e2718. doi:10.1002/smr.2718

[7] Michael Dubem Igbomezie, Phuong T Nguyen, and Davide Di Ruscio. 2024. When simplicity meets effectiveness: Detecting code comments coherence with word embeddings and LSTM. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 411–416. doi:10.1145/3661167.3661187

[8] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE Software* 20, 6 (2003), 35–39. doi:10.1109/MS.2003.1241364

[9] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic detection of outdated comments during code changes. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 154–163. doi:10.1109/COMPSAC.2018.00028

[10] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering* 49, 1 (2021), 1–23. doi:10.1109/TSE.2021.3138909

[11] Mary McHugh. 2012. Interrater reliability: The Kappa statistic. *Biochemia Medica* 22 (10 2012), 276–82. doi:10.11613/BM.2012.031

[12] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157. doi:10.1007/BF02295996

[13] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 427–435. doi:10.1609/aaai.v35i1.16119

[14] Fazle Rabbi and Md Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 371–375. doi:10.1145/3387904.3389286

[15] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *International Conference on Automated Software Engineering (ASE)*. 112–122. doi:10.1109/ASE.2017.8115624

[16] Martin P. Robillard, Deeksha M. Arya, Neil A. Ernst, Jin L. C. Guo, Maxime Lamothe, Mathieu Nassif, Nicole Novielli, Alexander Serebrenik, Igor Steinmacher, and Klaas-Jan Stol. 2024. Communicating Study Design Trade-offs in Software Engineering. *ACM Transactions on Software Engineering Methodologies (TOSEM)* 33, 5, Article 112 (June 2024), 10 pages. doi:10.1145/3649598

[17] Guoping Rong, Yongda Yu, Song Liu, Xin Tan, Tianyi Zhang, Haifeng Shen, and Jidong Hu. 2025. Code Comment Inconsistency Detection and Rectification Using a Large Language Model . In *International Conference on Software Engineering (ICSE)*. 432–443. doi:10.1109/ICSE55347.2025.00035

[18] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608. doi:10.1109/ICSE.2015.76

[19] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. 2020. Towards detecting inconsistent comments in Java source code automatically. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 65–69. doi:10.1109/SCAM51674.2020.00012

[20] Jiaxi Tan, Shikai Guo, Zijian Tao, Zhiguo Yang, and Hui Li. 2025. Just-In-Time Detection of Outdated Comments in Software Development by Jointly Reasoning. *IEEE Transactions on Consumer Electronics (TCE)* (2025). doi:10.1109/TCE.2025.3535632

[21] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the SIGOPS Symposium on Operating Systems Principles (SOSP)*. 145–158. doi:10.1145/1323293.1294276

[22] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 260–269. doi:10.1109/ICST.2012.106

[23] Gias Uddin and Martin P Robillard. 2015. How API documentation fails. *IEEE Software* 32, 4 (2015), 68–75. doi:10.1109/MS.2014.80

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)* 35 (2022), 24824–24837. doi:10.48550/arXiv.2201.11903

[25] Xiufeng Xu, Fuman Xie, Chenguang Zhu, Guangdong Bai, Sarfraz Khurshid, and Yi Li. 2025. Identifying Multi-Parameter Constraint Errors in Python Data Science Library API Documentations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/3728945

[26] Zhengkang Xu, Shikai Guo, Yumiao Wang, Rong Chen, Hui Li, Xiaochen Li, and He Jiang. 2024. Code Comment Inconsistency Detection Based on Confidence Learning. *IEEE Transactions on Software Engineering (TSE)* (2024). doi:10.1109/TSE.2024.3358489

[27] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. 2024. Leveraging Large Language Model to Assist Detecting Rust Code Comment Inconsistency. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 356–366. doi:10.1145/3691620.3695010

Received 2026-01-29; accepted 2026-06-25