Identifying Multi-parameter Constraint Errors in Python Data Science Library API Documentation

XIUFENG XU, Nanyang Technological University, Singapore
FUMAN XIE, University of Queensland, Australia
CHENGUANG ZHU, University of Texas at Austin, USA
GUANGDONG BAI, University of Queensland, Australia
SARFRAZ KHURSHID, University of Texas at Austin, USA
YI LI, Nanyang Technological University, Singapore

Modern AI- and Data-intensive software systems rely heavily on data science and machine learning libraries that provide essential algorithmic implementations and computational frameworks. These libraries expose complex APIs whose correct usage has to follow constraints among multiple interdependent parameters. Developers using these APIs are expected to learn about the constraints through the provided documentation and any discrepancy may lead to unexpected behaviors. However, maintaining correct and consistent multi-parameter constraints in API documentation remains a significant challenge for API compatibility and reliability. To address this challenge, we propose MPChecker for detecting inconsistencies between code and documentation, specifically focusing on multi-parameter constraints. MPChecker identifies these constraints at the code level by exploring execution paths through symbolic execution and further extracts corresponding constraints from documentation using large language models (LLMs). We propose a customized fuzzy constraint logic to reconcile the unpredictability of LLM outputs and detect logical inconsistencies between the code and documentation constraints. We collected and constructed two datasets from four popular data science libraries and evaluated MPChecker on them. The results demonstrate that MPChecker can effectively detect inconsistency issues with the precision of 92.8%. We further reported 14 detected inconsistency issues to the library developers, who have confirmed 11 issues at the time of writing.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Documentation**; • **Computing methodologies → Natural language processing**; **Information extraction**.

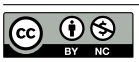
Additional Key Words and Phrases: API Documentation, Symbolic Execution, Python, LLM, Fuzzy Logic

ACM Reference Format:
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 Documentation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA068 (July 2025), 23 pages. https://doi.org/10.1145/3728945

1 Introduction

Machine learning (ML) and Artificial Intelligence (AI) have consistently garnered widespread attention, achieving remarkable breakthroughs in diverse domains including natural language processing, recommendation systems, autonomous vehicles, and robotics. Behind the rapid advancement of these transformative technologies, data science and machine learning libraries play a

Authors' Contact Information: Xiufeng Xu, Nanyang Technological University, Singapore, Singapore, XIUFENG001@e.ntu.edu.sg; Fuman Xie, University of Queensland, Brisbane, Australia, fuman.xie@uq.edu.au; Chenguang Zhu, University of Texas at Austin, Austin, USA, cgzhu@utexas.edu; Guangdong Bai, University of Queensland, Brisbane, Australia, g.bai@uq.edu.au; Sarfraz Khurshid, University of Texas at Austin, Austin, USA, khurshid@ece.utexas.edu; Yi Li, Nanyang Technological University, Singapore, Singapore, yi_li@ntu.edu.sg.



ACM 2994-970X/2025/7-ARTISSTA068
https://doi.org/10.1145/3728945

crucial role in AI and ML development. By providing extensive APIs for complex mathematical operations and algorithmic implementations, these libraries enable researchers and practitioners to focus on solving domain-specific problems rather than reimplementing fundamental algorithms.

A well-designed API documentation not only provides detailed descriptions of interfaces, including the purpose and range of parameters or attributes, returns, and exceptions thrown, but may also specify logical constraints or dependencies among multiple parameters. For data science and machine learning libraries, multi-parameter constraints are commonly mentioned in their API documentation and users of these libraries are expected to follow them closely when using the APIs. However, frequent version updates may lead to the documentation out of sync with the corresponding code, known as the **C**ode-**D**ocumentation **I**nconsistency (CDI) issue [14, 51]. Such CDI issues are particularly pronounced in data science libraries. On one hand, the underlying mathematical models of DS/ML libraries inherently come with various constraints, such as *"a model X can only be chosen when a parameter Y is provided"*, and incorrect parameter configurations not satisfying their constraints may lead to unexpected outcomes. On the other hand, the number of parameters/attributes of these libraries can be significantly more than a typical library API, sometimes over a few dozen. Therefore, it is unrealistic to track all parameter constraints manually.

Detecting errors in multi-parameter constraints from Python API documentation is challenging for several reasons. (1) The quality of API documentation varies and lacks standardized writing guidelines. Some API documentation uses ambiguous language, contains typos, and may not follow a consistent styling guide. This makes simple rule-based pattern-matching approaches ineffective. (2) Existing approaches [37, 53] for detecting documentation errors focus on a single parameter only: e.g., checking whether the information provided on parameter ranges, nullness, and identifier names is correct. It is more challenging to extract multi-parameter constraints precisely from free-style descriptions written in natural languages. (3) For the same reason, a semantic-aware code analysis approach is essential, as logical relations among multiple parameters cannot be easily identified through purely syntactic analysis. The challenge is further compounded by Python's dynamic nature, where variable types, attributes, and behaviors can change at runtime.

To detect multi-parameter constraint inconsistencies from data science library documentation, we propose an automated tool MPChecker. MPChecker identifies inconsistencies between API documentation and the corresponding library code by combining symbolic execution-based program analysis techniques with constraint extraction methods powered by large language models (LLMs). We first extract multi-parameter constraints from documentation (a.k.a. *doc-constraints*), leveraging the powerful natural language understanding capability of LLMs. We incorporate a few optimizations, such as Chain of Thought (CoT) [68] and few-shot learning to improve the accuracy of constraint extraction. Then we use dynamic symbolic execution to collect all path constraints from the corresponding Python source code. The symbolic path constraints (a.k.a. *code-constraints*) capture the real constraints that the parameters have to follow according to the library code, which are then used to evaluate the correctness of the *doc-constraints*.

Then, in order to mitigate minor discrepancies that may arise from the *doc-constraints* extracted by LLMs, we design and implement a *Fuzzy Constraint Logic* (FCL) framework to estimate how logically consistent a *doc-constraint* is with a set of given *code-constraints*. Intuitively, in the absence of LLM-induced unpredictability, a *doc-constraint* must be evaluated as true under the assumption of *code-constraints*. Through fuzzy constraint satisfaction, we can accommodate many *nearly-correct* constraints produced by LLMs and thus improve the accuracy of the overall approach.

Contributions. Our work aims to integrate precise symbolic reasoning with the inherently fuzzy outputs of large language models. To summarize, we make the following contributions.

(1) We proposed an automated multi-parameter code-documentation inconsistency detection technique and developed an end-to-end command-line tool called MPChecker. Existing techniques in the same area are only designed to handle single parameter inconsistencies, without considering inter-parameter constraints.
(2) We introduced a customized fuzzy constraint satisfaction framework to mitigate the uncertainties introduced by LLM outputs. We provide a theoretical derivation of the membership function based on constraint similarity.
(3) We constructed a documentation constraint dataset comprising 72 real-world constraints sourced from widely used data science libraries, and derived a mutation-based inconsistency dataset with 216 constraints. Our dataset and tool implementation are made available online: https://github.com/ParsifalXu/MPChecker
(4) We evaluated our tool on four real-world popular data science libraries. We reported 14 inconsistency issues discovered by MPChecker to the developers, who have confirmed 11 inconsistencies at the time of writing.

2 Background

In this section, we review the essential terminology and background necessary for understanding the remainder of the paper.

2.1 Multi-Parameter Constraints

We use two examples to illustrate inconsistencies between API documentation and the corresponding code caused by multi-parameter interdependence. Both of them come from open-source Python data science libraries and were successfully detected by MPChecker. In general, there are two types of constraints found in the API documentation. (1) An *explicit constraint* clearly specifies the logical relationship among two or more interrelated parameters. (2) An *implicit constraint* is an unstated or indirectly implied relationship among two or more interrelated parameters, where the constraint is inferred through contexts or convention rather than explicitly specified.

2.1.1 Example 1: Explicit Constraint. The first example comes from `statsmodels` [63], which provides a complement to `scipy` for statistical computations including descriptive statistics and estimation and inference for statistical models. `Statsmodels` has more than 10K stars on GitHub and is actively maintained. Figure 1 illustrates an inconsistency caused by an explicit constraint from the class *AutoReg*. The relevant portions for the *doc-* and *code-constraints* are highlighted. As mentioned in the documentation of `deterministic`, the trigger condition for the warning is that "trend is not n, **and** seasonal is not False". However, it is apparent that the *code-constraint* for `trend` and `seasonal` (to be used together correctly and avoid any warning) implemented is **or** instead of **and**. One way to fix the documentation is to change "and" to "or".

2.1.2 Example 2: Implicit Constraint. The second example comes from `scikit-learn` [59], which is a widely-used (more than 60K stars on GitHub) open-source ML library in Python, designed to offer simple and efficient tools for data mining and data analysis. Figure 2 displays an inconsistency caused by an implicit constraint from the class *SpectralClustering*. It is evident that the highlighted part of the documentation only explicitly mentions one parameter *affinity*, omitting the subject "gamma". More importantly, "ignore" is not a specific identifier or value but rather a description of the program logic—if the parameter *affinity* is set to *nearest_neighbors*, then the parameter *gamma* will not be used. Whereas, above constraint does not faithfully reflect the behavior implemented in code. According to the code snippet, "gamma" is not only ignored within the *nearest_neighbors* branch, but also ignored within the *precomputed_nearest_neighbors* and *precomputed* branches. This indicates that the constraint is inaccurate and demonstrates a form of inconsistency.

Constraint description of `trend` and `seasonal` in class `AutoReg`

> **deterministic:** DeterministicProcess
A deterministic process. If provided, trend and seasonal are ignored. A warning is raised if trend is not "n" and seasonal is not False.

Corresponding code snippet in class `AutoReg`

```
class AutoReg(tsa_model.TimeSeriesModel):
    def __init__(...):
        if deterministic is not None and (self.trend != "n" or self.seasonal):
            warnings.warn('When using deterministic, trend must be "n"
                and seasonal must be False.', SpecificationWarning, stacklevel=2)
```

Fig. 1. Examples of an explicit constraint from `Statsmodels`.

For this type of implicit constraint, traditional pattern-based approaches are not able to extract the *doc-constraint* correctly, thus fail to detect the inconsistencies. To solve this issue, we design a customized constraint that incorporates fuzzy words, and adopt few-shot learning to teach LLMs how to generate such constraints (details in Section 3.2.2). In this case, the *doc-constraint* should be *"(affinity = "nearest_neighbors") → (ignore(gamma))"*, where a special predicate *"ignore(x)"* is used to indicate that a parameter x is ignored (see Section 3.3.1).

2.2 Fuzzy Logic and Fuzzy Constraint Satisfaction

Unlike traditional Boolean logic, fuzzy logic [31] is a multi-valued logic that allows for values between 0 and 1 to represent varying degrees of truth, where 0 represents absolute false, and 1 represents absolute true. The human brain can process vague statements or claims that involve uncertainties or subjective judgments, such as "the weather is hot", "that man runs so fast", or "she is beautiful". Unlike computers, humans possess common sense, allowing them to reason effectively in situations where things are only partially true. Fuzzy logic is primarily used to model uncertainty and vagueness, making it highly applicable in real-world scenarios where precision may be difficult or impossible to achieve.

A traditional constraint satisfaction problem (CSP) [18] requires all constraints to be fully satisfied. Constraints are either completely satisfied or unsatisfied, which is why these strict, non-fuzzy constraints are referred to as "*crisp constraints*". An extension of CSP, known as soft CSP [39, 58], introduces a distinction between hard constraints and soft constraints. Hard constraints must be absolutely satisfied, while soft constraints are typically assigned a weight or priority, allowing for lower-weighted constraints to be only partially satisfied or even unsatisfied under certain conditions during problem-solving. Another extension is Fuzzy CSP [56], which differs from soft constraints in that it incorporates fuzzy logic and allows each constraint to be "partially satisfied" to a degree, quantified by a "satisfaction degree". This satisfaction degree usually ranges from 0 to 1, indicating the extent to which a constraint is fulfilled. The goal in fuzzy constraint satisfaction is to find a solution that maximizes satisfaction, rather than strictly satisfying all constraints.

3 Methodology

In this section, we define the issue of code-documentation inconsistency caused by multi-parameter constraints and provide a detailed description of our approach. An API documentation error is

Constraint description of `gamma` and `affinity` in class `SpectralClustering`

> **gamma** : float, default=10
Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. **Ignored for affinity="nearest_neighbors".**

Corresponding code snippet in class `SpectralClustering`

```
class SpectralClustering(ClusterMixin, BaseEstimator):
    def fit(self, X, y=None):
        if self.affinity == "nearest_neighbors":
            ...
        elif self.affinity == "precomputed_nearest_neighbors":
            ...
        elif self.affinity == "precomputed":
            ...
        else:
            params = self.kernel_params
            if params is None:
                params = {}
            if not callable(self.affinity):
            params["gamma"] = self.gamma
                params["degree"] = self.degree
                params["coef0"] = self.coef0
```

Fig. 2. Examples of implicit constraint from `Scikit-learn`.

an inconsistency between the library source code and its API documentation. Multi-parameter constraints refer to conditional dependency relationships that exist among multiple parameters within functions or classes. If a constraint is never violated across all execution paths in the code, it is considered as a benign constraint, or it indicates a potential documentation error. According to literature [65, 79, 80], API documentation inconsistency can be categorized into two types: incorrectness and incompleteness. Incorrectness refers to cases where the documentation describes behavior that is not implemented in the code, while incompleteness arises when certain code behaviors are not reflected in the documentation. Typically, incorrectness issues are considered more critical than incompleteness.

In addition, when it comes to constraint extraction, compared to single-parameter constraints, we need to classify the multi-parameter constraint extraction problem into two types, as discussed in Section 2.1, 1) explicit constraint and 2) implicit constraint.

MPChecker aims to accurately extract multi-parameter constraints from API documentation and detect both types of inconsistency. As the architecture depicted in Figure 3, we have designed a three-phase workflow comprising the **1) Data Preprocessing; 2) Constraint Extraction; 3) Inconsistency Detection**. During the preprocessing phase, we separate the code and documentation within the project. MPChecker will then automatically rewrite each function to be compatible with the symbolic execution tool. This includes replacing advanced Python syntax that the tool cannot handle with simpler constructs and symbolizing external function calls, such as replacing ternary conditional expressions with if-else statements. These modifications do not alter the path constraints of the original program. In the constraint extraction and expression generation phase,

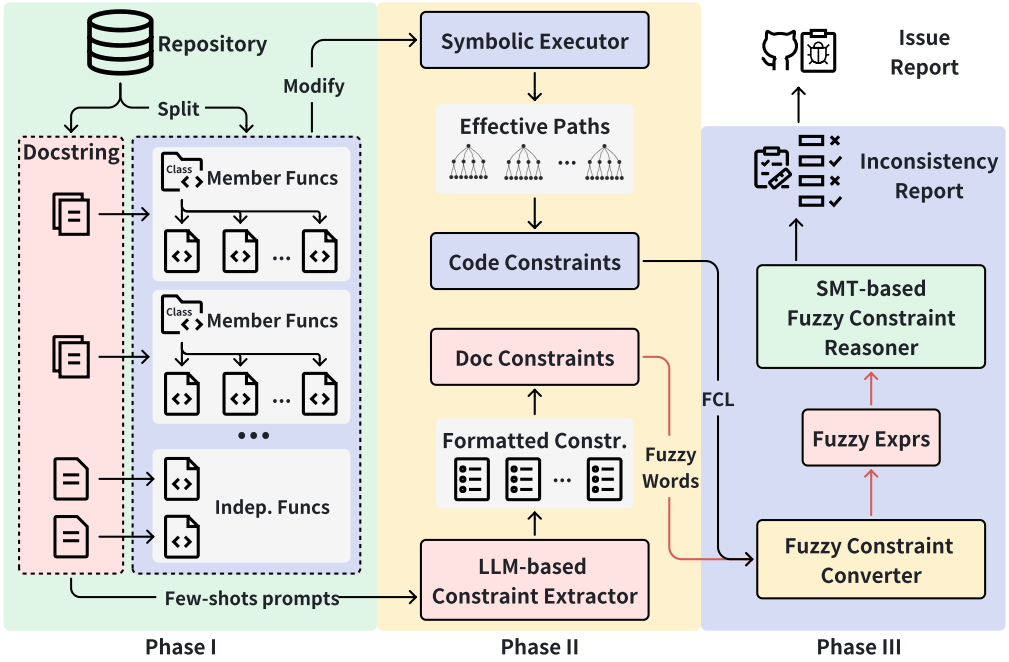


Fig. 3. The architectural overview of MPChecker.

on the one hand, we leverage large language models to extract constraints in a specific format from the documentation. On the other hand, the symbolic execution tool dynamically analyzes the code and solves the constraint paths. Those constraints are then converted into expressions that can be processed by the SMT solver. In the fuzzy constraint checking phase (Phase III), a constraint checker with SMT solver and fuzzy constraint reasoner performs comprehensive reasoning to detect inconsistencies. It is worth noting that we propose and implement an extended fuzzy constraint satisfaction to mitigate the hallucination issues often introduced by large language models, and reduce the risk of false positives and missed detections.

3.1 Preprocessing

In this step, we will discuss the details of separating the documentation and corresponding code from the project and the specifics of preprocessing the documentation content.

In modern data science libraries, documentation is typically auto-generated using Sphinx, a tool that can automatically create HTML documentation from Python code. Sphinx supports various docstring styles, with Google style and NumPy style being commonly used. Figures 4 respectively display docstring examples of two different styles from Sphinx official website [61, 62]. Google-style docstrings use a clear and concise format with a minimalistic structure. It divides the docstring into sections like **Args**, **Attributes**, etc., with each section using plain indentation. Similarly, Numpy-style docstrings organize sections more rigidly. Sections are divided by using **Parameters**, **Attributes**, etc. with horizontal dash lines "- - -" under the section header. The number of dashes is the same as the number of letters in the section header. Regardless of the style used, the docstring is normally placed at the beginning inside its corresponding class or function.

Numpy Style Docstrings

```python
class ExampleNumpyStyle():
  """Exceptions are documented
  Parameters
  ----------
  msg : str
    Human readable string describing the
        exception.
  code : obj:`int`, optional
    Numeric error code.
  Attributes
  ----------
  msg : str
    Human readable string describing the
        exception.
  code : int
    Numeric error code.
  """
  def __init__(self, msg, code):
    self.msg = msg
    self.code = code
```

Google Style Docstrings

```python
class ExampleGoogleStyle():
  """Exceptions are documented
  Note:
    Do not include the `self` parameter
    in the ``Args`` section.

  Args:
    msg (str): Human readable string
        describing the exception.
    code (:obj:`int`, optional): Error
        code.

  Attributes:
    msg (str): Human readable string
        describing the exception.
    code (int): Exception error code
  """
  def __init__(self, msg, code):
    self.msg = msg
    self.code = code
```

Fig. 4. Example of two docstring styles

After downloading the project, our tool first converts every Python file from the project into an Abstract Syntax Tree (AST) and isolates the classes and independent functions. This paper focuses on the CDI issue, so in this step, we filter out code without documentation and separately extract the code and documentation from the remaining code. Since Python supports object-oriented programming but current symbolic execution tools have limited support for classes, we have to limit our experimental units to functions. For independent functions, the scope of constraints in the documentation usually applies within the function itself. For classes, however, the constraints cover the entire class, including each member function. Therefore, we create a new directory for every class and independent function, with member function directories placed within their corresponding class directories to maintain structural consistency. If the member function has its own documentation, it will also be retained.

To help the LLM better focus on the constraints between parameters and reduce the occurrence of erroneous constraints, we retained parameters or attributes and their corresponding descriptions in the form of key-value pairs, based on the two aforementioned docstring styles. We subsequently applied a rule-based heuristic approach to retain documentation that potentially contain constraints and to discard the rest. For instance, if none of the parameters or attributes appear in other descriptions, this indicates the absence of multi-parameter constraints in that documentation.

3.2 Constraint Extraction

We now specially explain how to extract path constraints from code and convert them into expressions which are solvable by SMT solver, as well as how to use LLM to extract constraints from documentation and transform them into expressions containing fuzzy words.

3.2.1 Code Constraint Expression Extraction. The goal of MPChecker is to verify whether the constraints between multiple parameters in documentation align with the logic during actual code execution. This requires our tool to understand and analyze deeper constraint relationships. Therefore, we employ symbolic execution to capture as many path conditions as possible and precisely handle complex paths and constraints.

We modified current advanced dynamic symbolic execution tools [16, 19, 48–50] for path exploration. Unfortunately, supporting dynamic languages like Python is more challenging compared to symbolic execution tools designed for static languages such as Java and C. Despite Python's rapid evolution, symbolic execution tools specifically designed for Python have developed slowly, struggling to keep pace with the growing new syntax and features. This forces us to make reasonable modifications to the source code extracted directly from repositories. However, these modifications must not alter the path constraints of the original code; they should be equivalent code transformations that do not affect path exploration. We mainly made the following modifications:

(1) Current Python symbol execution tool can not solve class directly. Therefore, it is necessary to split the class into functions (i.e. member functions). The corresponding member variables also need to be changed and used as symbolic inputs.
(2) Replace complex structures and operations, such as lists and dictionaries, that are difficult to handle and do not affect the path, as well as external function calls that may cause path explosion, with symbolic inputs.
(3) Replace the handling of exceptions and warnings that do not affect the path with *return*.
(4) Add a fixed format of *return* statement to capture concrete values of potential symbols.
(5) Equivalent code implementation replacement to avoid being unable to find useful path constraints due to poor support for some advanced syntax. For example, replace ternary operator to conventional if-else statement.

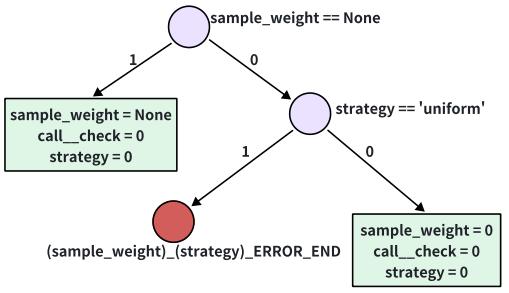
In the limit, MPChecker strives to explore all feasible paths in a Python function by following these processes: 1) Running the function with specific input to trace a path through the control flow of the function; 2) Symbolic executing the path to determine how its conditions depend on the function's input parameters; 3) Utilizing Z3 to generate new parameter values that guide the function toward paths that haven't been covered yet.

Although MPChecker supports a certain level of external function call analysis, in complex real-world code, an external function call often corresponds to extra more function calls, leading to path explosion. Furthermore, documentation constraints are usually handled within the target function, so we still prefer not to introduce external function calls and to focus the analysis within the target function. Additionally, similar to the current concolic symbolic execution tools for Python, MPChecker does not yet provide strong support for theorem of strings. Thus, during the actual execution process, we replace the string with a unique large number, which does not affect the exploration of condition constraints.

We will use an example depicted in Figure 5 to illustrate the entire extraction phase, containing a simplified original source code and modified code from a popular data science project `scikit-learn`, and its corresponding path constraints. The *fit* function is a member function within a class, and thus member variables such as "self.strategy" also exist within the code. We also modified "None" as a string to make it easier to be captured, since it is represented as a number 0 during symbolic execution. Our tool first modifies the code and replaces exception handling and external function calls with symbolic inputs, marked as "ERROR_END" and "call_", respectively. For those paths whose final states are "ERROR_END", the final results of the conjunction of the documentation constraint and these paths will be negated during reasoning phase.

Original source code

```
def fit(self, sample_weight):
  if sample_weight is not None and self.
       strategy == "uniform":
    raise ValueError("Warning Info")
  if sample_weight is not None:
    sample_weight = _check_sample_weight(
         sample_weight, X)
```



Modified source code

```
def fit(sample_weight, strategy, call__check_sample_weight):
  if sample_weight != 'None' and strategy == 'uniform':
    return '(sample_weight)_(strategy)_ERROR_END'
  if sample_weight != 'None':
    sample_weight = call__check_sample_weight
  return (f'(sample_weight = {sample_weight}) ^ (call__check_sample_weight =
                    {call__check_sample_weight}) ^ (strategy = {strategy})')
```

Fig. 5. Extracting constraint from code

3.2.2 Documentation Constraint Expression Extraction. In this step, we extract constraints from Python documentation by applying LLMs. Since Python documentation can vary in quality and may contain informal writing [52], the important task is to understand the parameter information within the documentation. To achieve this, we resort to SOTA LLMs. Given Python documentation as input, the LLM is asked to first extract constraint-related sentences and then output them in a standard logical expression format. This includes two steps, model selection and prompt design.

Model Selection. We adopt GPT-4, which is pretrained on a diverse corpus and shows excellent performance in natural language understanding. Based on our preliminary study, GPT-4's performance stands out compared to Gemini-1.5 [22] and LLaMA-3 [40] due to its ability to capture details, and it is also well-acquainted with the context of code documentation [21].

Prompt Design. Because the constraint extraction task is relatively complex and can be broken down into clear steps, we apply the chain-of-thought approach [67], which has been widely proven effective in improving GPT-based model performance. We first divide the prompt task into two steps, document input and constraint extraction. Figure 6 shows the structure and some details of the used prompt. Below, we detail our prompt mechanism for each step.

Document Input Prompt. We observe that some documentation may be too lengthy to provide to GPT-4 in a single input, considering that GPT-4 has a maximum token length limit of 8,192 tokens [45]. We also find that LLMs exhibit lower performance when dealing with long and complex text inputs as noted in previous research [24, 27]. Thus, we decide to segment the lengthy documents into smaller sections. To determine a heuristic chunk size, we randomly select ten lengthy Python documentation files, split them into chunks of varying word lengths, and use these as inputs for GPT-4. We then evaluate the constraint extraction task performance of GPT-4 based on these inputs to determine which chunk size yields better results. Based on our findings, we decide to standardize the chunk size to 1,500 words (around 2,048 tokens [46]). We also input the parameter list obtained

in Section 3.1 into GPT-4 to help model better recognize the information related to parameters. The details of the document input prompt are shown in Prompt 1 in Figure 6.

Constraint Extraction Prompt. For the constraint extraction task, our prompt is divided into three parts to guide GPT-4 in recognizing text related to constraints in the original documentation, and then, based on that text, to generate a formatted logical expression of the constraint.

The first part involves defining the logical symbols that can be used in the logical format, including implication, negation NOT, logical AND, logical OR, and also defining parentheses to indicate the precedence of logical expressions.

The second part arises from our preliminary study, in which we observed that some Python documentation uses vague terms such as “override”, “specify”, “have an effect”, “no effect”, “significant”, and “ignore” when mentioning constraints related to parameters. To preserve as much detail as possible from the documentation, we design prompts to guide GPT-4 so that if text related to parameter constraints contains vague keywords, these keywords should be retained in the final logical expression.

In the third part, to ensure that the format of the logical expression in GPT-4’s output is consistent each time and convenient to process, we apply in-context learning techniques that widely used in previous works [41, 55] to enable GPT-based models to handle tasks specific to a domain. We include four examples that contain pairs of original constraint-related sentences selected from Python documentation and their corresponding logical expression constraints.

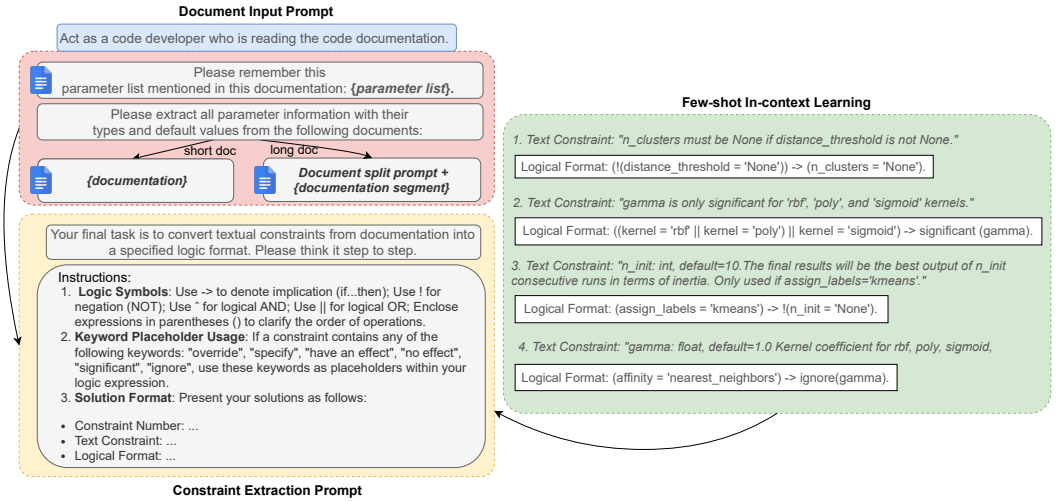


Fig. 6. Prompt structure for constraints extraction

3.3 Inconsistency Detection

In the second phase, we extracted constraints from both documentation and code. While *code-constraints* are deterministic in nature, *doc-constraints* inherently contain uncertainties stemming from two main sources. First, there are *implicit constraints* arising from vague or incomplete descriptions, which we address by defining fuzzy words to extend them into soft constraints. Second, we encounter uncertainties introduced by generative models’ limited reasoning capabilities and unavoidable hallucination issues, for which no validator exists to definitively determine the correctness of generated constraints. To address this challenging scenario, we proposed a customized

fuzzy constraint logic to mitigate such vagueness. With the help of fuzzy words and fuzzy constraint logic, our converter can effectively handle both explicit and implicit constraints. The converter ultimately produces fuzzy expressions, which are then processed by a reasoner based on the z3 SMT (Satisfiability Modulo Theories) solver to detect inconsistencies.

The reasoner offers two strategies, from relaxed to strict, to identify inconsistencies from the perspectives of satisfiability and equivalence. Given a *doc-constraint* c and a *code-constraint* set P containing a group of path constraints p, the detection strategies are defined as follows:

- **Unsatisfiability** checking determines whether the *doc-constraint* c is unsatisfiable under all path constraints. If the conjunction of c and every path constraint p is unsatisfiable, it indicates a contradictory inconsistency, meaning that under all possible execution paths, the code violates the constraint stipulated in the documentation.

$$\forall p \in P, \neg(c \wedge p) \tag{1}$$

- **Nonequivalence** checking determines whether the *doc-constraint* c and *code-constraints* are logically equivalent. If equivalence holds only under specific conditions, a behavioral inconsistency may arise, implying that the constraints implemented in the code are not fully equivalent to the documentation.

$$\exists p \in P, \neg(c \Leftrightarrow p) \tag{2}$$

For constraints containing sub-constraints that do not exist in the code logic, we employ a heuristic approach to provide suggestions. Specifically, when a constraint mentioned in the documentation is not present in the corresponding code logic, we issue a warning and label this constraint as a potential weak constraint to prompt further investigation by the user.

3.3.1 Fuzzy Words. The example from Section 2.1.2 illustrates a very typical implicit constraint with fuzzy words, where part of constraint is clearly defined while others remain uncertain. We introduce a series of fuzzy words to help LLM extract constraints better. These fuzzy words frequently appear in documentation but don't represent specific values, making it challenging for the LLM to extract them directly. We generally categorize these fuzzy words into two types: *existence* and *non-existence*. In fuzzy words, *non-existence* includes "ignore", "no effect", "unused", "override", indicating that a parameter either is unused or does not exist within code segments where other conditions are met. Similarly, *existence* includes "specify", "have an effect", "exist", "significant", indicating that the parameter is used or exists when other conditions are met.

We implement several specialized predicates to evaluate such implicit constraints. MPCheck will first trace the target variable's define-use chain (DU-chain) [25, 30] and then check if the definition and usage of it are existed or not under a specific program path. For instance, "`exist(x)`" will check if the definition and usage of a variable `x` are existed in the program path with other explicit conditions. If found, it will return `True`; otherwise, it will return `False`. Similarly, "`ignore(x)`" will check whether the definition and usage of `x` are absent under the program path with given explicit conditions. It can be further extended to check weak *doc-constraints*. Like the example in Section 2.1.2, *gamma* will be ignored not only when "`affinity=nearest_neighbors`" but also ignored when "`affinity=precomputed_nearest_neighbors`" as well as "`affinity=precomputed`".

3.3.2 Fuzzy Constraint Satisfaction. While we have employed several strategies in Phase II to maximize LLM's understanding of constraints and restrict randomness in outputs, inaccurate extraction is still unavoidable. The main reasons are threefold: (1) typos inevitably occur when developers write documentation; (2) the hallucination issues inherent to black-box generative models; and (3) the intrinsic ambiguity in natural languages. This implies that correct documentation

$$
\begin{aligned}
c \in \mathit{Constraint} &::= e \mid \neg c \mid c \vee c \mid c \wedge c \\
e \in \mathit{Expression} &::= p \bowtie v \\
p \in \mathit{Parameter} &::= \mathit{char}, \{\mathit{char} \mid \mathit{digit}\} \\
\bowtie \in \mathit{Operator} &::= < \mid > \mid <= \mid >= \mid = \mid \texttt{!=} \\
v \in \mathit{Value} &::= \mathit{string} \mid \mathit{number} \mid \mathit{bool}
\end{aligned}
$$

Fig. 7. Extended Backus-Naur form for multi-parameter constraint.

descriptions can generate incorrect *doc-constraints*, and incorrect documentation descriptions can also have the chance to generate correct *doc-constraints*.

In the absence of LLM unpredictability, detecting CDI issues is a crisp constraint satisfaction problem (CSP), deciding whether a *doc-constraint* is consistent with the actual code implementation. Nevertheless, due to minor errors introduced by LLMs, such as a single letter being wrongly spelled in a parameter name, or a comparison operator being reversed, e.g., writing "<" instead of ">", a *doc-constraint* can be mistakenly identified as inconsistent when it is actually correct.

To address this, we proposed a customized fuzzy constraint logic that reconciles such unpredictability. In a traditional fuzzy constraint [31, 56], a membership function assigns a degree of satisfaction (ranging from 0 to 1) to each possible variable value. It enables partial fulfillment of a condition, with satisfaction measured on a continuous scale. In our case, a constraint needs to be measured on a new scale, assessing "how likely" the extracted *doc-constraint* conforms to the *code-constraints*. Therefore, we introduced a unique similarity computation which serves as the membership function.

Figure 7 shows an EBNF grammar for our multi-parameter constraints. A multi-parameter constraint is a combination and nesting of binary expressions and Boolean operators, which can be viewed as a complete binary tree where leaf nodes are binary expressions over single parameters and non-leaf nodes are logical operators connecting them. Without loss of generality, we only keep negation, conjunction, and disjunction in the constraints; logical relations such as implications can be simplified accordingly. The fuzziness of a constraint is defined with respect to a set of *environment expressions*, facts that are known to hold (with a truth value of 1). In other words, the instantiation of a specific tree structure and nodes is a constraint c evaluated against a set of expressions $\{e_1, e_2, \ldots, e_n\}$. Next, we define the membership function of our fuzzy constraint logic through a few similarity functions.

Definition 3.1 (Expression Similarity). The similarity between two expressions e_1 and e_2 is defined as,

$$
\sigma(e_1, e_2) = \alpha * \left(1 - \frac{LD(p_1, p_2)}{\max(|p_1|, |p_2|)}\right) + \beta * \left(\frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\| \|\delta_{\bowtie_2}\|}\right) + \alpha * \left(1 - \frac{LD(v_1, v_2)}{\max(|v_1|, |v_2|)}\right), \tag{3}
$$

where α and β denote the relative weights, p, $\bowtie$, and v are parameter, operator, and value, respectively, $|p|$ and $|v|$ denotes the length of p and v, $\|\delta_{\bowtie}\|$ denotes the magnitudes (or Euclidean norms) of the vector $\delta_{\bowtie}$.

The similarity between two expressions are considered separately for the parameters, operators, and values appeared in the expressions. Both p and v can be treated as texts, therefore, Levenshtien Distance (a.k.a. edit distance) is used to represent their similarity. The normalized Levenshtien Distance (NLD) is given in Eq. (4), where s denotes strings (p or v) and $|s|$ denotes the length of it.

$$\eta(s_1, s_2) = NLD = 1 - \frac{LD(s_1, s_2)}{\max(|s_1|, |s_2|)} \tag{4}$$

$$\delta_{\bowtie} = (C, E, G, L, N), \text{where } C, E, G, L, N \in \{0, 1\} \tag{5}$$

$$cos\theta(\bowtie_1, \bowtie_2) = \frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\| \|\delta_{\bowtie_2}\|} \tag{6}$$

As illustrated in Eq. (5), we design an operator embedding across five key dimensions: **C**omparison, **E**quality, **G**reater than, **L**ess than, and **N**egativity. This way, we may calculate the similarity between two operators by simply calculating the cosine similarity between two vectors. The result is highly intuitive. For example, with $\delta_{<} = (1, 0, 0, 1, 0)$, $\delta_{>} = (1, 0, 1, 0, 0)$, and $\delta_{<=} = (1, 1, 0, 1, 0)$, the similarity between "<" and ">" is 0.5, while the similarity between "<" and "<=" is 0.82.

The weight of operator similarity is set as β such that the weights of operators, values, and parameters within a given experssion should sum to one. Thus, we have $\alpha = \frac{1-\beta}{2}$ and the similarity σ of two single parameter expressions (i.e., atomic constraint) can be calculated according to Eq. (3).

Definition 3.2 (Constraint Similarity). Let c be a constraint and $\Phi = \{e_i | i = 1, \ldots, n\}$ be a set of environment expressions assumed to hold true. The similarity of c against Φ is given by the following set of calculations.

$$\rho(c, \Phi) = \begin{cases} \arg\max_{e_i \in \Phi} \sigma(e, e_i), & \text{if } c \text{ is an expression } e \\ 1 - \sigma(c', \Phi), & \text{if } c = \neg c' \\ \min\{\sigma(c_1, \Phi), \sigma(c_2, \Phi)\}, & \text{if } c = c_1 \wedge c_2 \\ \max\{\sigma(c_1, \Phi), \sigma(c_2, \Phi)\}, & \text{if } c = c_1 \vee c_2 \end{cases} \tag{7}$$

Consider an atomic constraint with a single expression; its similarity to Φ associated with a set of *environment expressions* can be represented by the maximum expression similarity among all expressions within Φ. Based on the *conjunctive combination principle* [72], when combining two constraints using a conjunction, their degree of joint similarity ρ should be represented by the minimum similarity between them. Similarly, based on the *disjunctive combination principle* [72], when they are combined with a disjunction, the maximum similarity should be used. For negation, the complementary similarity is used.

Definition 3.3 (Membership Function for Fuzzy Constraint Satisfaction). Constraint similarity serves as the membership function μ_Ω, quantifying the degree to which a given constraint ϵ is consistent with the code, which is represented as a set Ω of path constraints ω:

$$\mu_\Omega(\epsilon) = \rho(\epsilon, \Phi_\Omega) \cdot \epsilon[e \mapsto e_{\Phi_\Omega}] \tag{8}$$

where Φ_Ω denotes the set of expressions aggregated from all the path constraints in Ω, and $\epsilon[e \mapsto e_{\Phi_\Omega}]$ is a rewrite of ϵ, where each expression e has been replaced by its cloest counterpart from Φ_Ω.

The inconsistency between the modified constraint $\epsilon[e \mapsto e_{\Phi_\Omega}]$ and Ω is then evaluated (according to Eq. (1) or Eq. (2)), yielding a binary result (`True` or `False`). To enable the probabilistic interpretation, a linear transformation ensures complementary probabilities. For instance, "0.7·False = 0.3·True", indicating a 70% probability of inconsistency or a 30% probability of consistency.

3.3.3 Constraint Similarity Threshold. LLMs demonstrate a great potential in constraint extraction, yet they still encounter errors such as using incorrect parameter names or values and introducing non-existent constraints. To reduce false positives from these inevitable issues, we set a constraint similarity threshold of 0.85. This is based on the observation that a high constraint similarity (>0.85) indicates a high likelihood of misinterpretation or conflation by the LLM.

For instance, in `scikit-learn`, the documentation of `LinearSVC` states: "*If n_samples < n_features and optimizer supports chosen loss, multi_class and penalty, then dual will be set to True*". However, the extracted constraint is "`(samples<features)`∧`(dual=True)`", where two parameters are mistakenly mapped to similar names. Their constraint similarity is 0.86, exceeding the threshold, leading MPChecker to discard the result. Moreover, in most cases where the expression contains only a single parameter, the threshold exhibits stronger filtering capability.

However, setting a threshold cannot entirely exclude all false positives, as there is no definitive rule to ascertain whether an error originates from the documentation or the LLM. Another example from `scikit-learn` illustrates this limitation: the documentation of `estimator_` states: *"The child estimator template used to create the collection of fitted sub-estimators"*. Yet, the extracted constraint "`(estimator_=child_estimator_template)` ∧ `(collection=fitted_sub_estimators)`" has a constraint similarity of 0.67, which is below the threshold, leading MPChecker to accept the result.

4 Evaluation

This section describes our evaluation of MPChecker. We first present our research questions, then detail the experiment setup and evaluation subjects. Finally, we analyze our experimental results and answer each research question. Our evaluation was guided by the following research questions:

(1) **RQ1:** How accurate is MPChecker in extracting constraints from API documentation?
(2) **RQ2:** How effective is MPChecker in detecting errors related to multi-parameter constraints in API documentation?
(3) **RQ3:** How effective can MPChecker detect unknown inconsistency issues?

4.1 Experiment Setup

4.1.1 Dataset. There is currently no well-established dataset specifically focusing on multi-parameter API documentation errors. To better evaluate the effectiveness of our tool, we constructed two datasets: a constraint dataset and an inconsistency dataset.

Constraint Dataset. We constructed a dataset containing 72 constraints from 4 popular open-source data science libraries with well-maintained documentation, including `scikit-learn`, `scipy`, `numpy`, and `pandas`. The constraints were gathered by analyzing commits from each GitHub repository, focusing on developers' modifications to documentation related to multi-parameter constraints. To streamline this process, we developed an automated script to collect all documentation-related commits and identify changes within parameter (including attribute) descriptions, adhering to two distinct docstring styles (see details in Figure 4). By mapping parameter names to their descriptions, it then cross-checks if any parameter names appear within others' descriptions to keep potential constraint-related documentation. Approximately 90% to 95% of irrelevant commits are excluded, leaving a smaller subset of commits that may contain constraints. Additionally, we perform a random sampling of the excluded commits to assess and minimize the impact of this heuristic. Despite the approach's proven effectiveness, the remaining subset still contains a substantial number of submissions. Each of the four repositories has over 30,000 commits, requiring manual verification of approximately 1,500 commits per repository to further identify multi-parameter constraints. For each constraint, we log its source information (repository name, SHA, file path, etc.) and retain the code file, enabling swift extraction of documentation and code for reproducibility. In some cases,

Table 1. Data science libraries used in the experiments

Project	class	class w/ doc	func	func w/ doc	KLOC	avg. params	#Stars
scikit-learn	878	306	9,332	1535	400.1	1.42	61.8K
pandas	2,211	102	28,880	1432	620.6	1.14	45.2K
scipy	2,570	142	22,059	1705	517.9	1.30	13.6K
numpy	2,000	51	12,618	902	276.3	0.83	29.4K
keras	1,370	254	9,877	724	218.5	1.42	62.9K
dask	284	26	6,914	368	157.5	1.33	13.1K
statsmodels	2,184	273	11,590	1,894	424.6	1.32	10,6K

the condition enforcing the constraint is located not in the current function but in a function it calls, leading to a mismatch between documentation and code. To address this, we record such mismatches and relocate the constraint description to the function where the check actually occurs.

Inconsistency Dataset. Based on constraint dataset, we constructed an inconsistency dataset comprising 126 multi-parameter constraints that lead to code-documentation inconsistencies. We analyzed around 20 resolved GitHub issues related to multi-parameter constraints and identified eight common patterns that may cause CDI: (1) Parameter name change; (2) Value Change; (3) Logic Change; (4) Remove Parameter; (5) Add Constraints; (6) Remove Constraints; (7) Missing Documentation; (8) Modify Description. To better evaluate the capabilities of our tool, we applied these eight patterns to mutate the dataset based on the *Constraint Dataset*. For each constraint, we applied two types of modifications, resulting in an inconsistency dataset containing 216 constraints. We feed the mutated constraints and the original constraints into an SMT solver to verify if the mutations violate the original constraints. We also manually inspected each of them to ensure the constraint was inconsistent. It is important to note that modifying a correct constraint does not necessarily turn it into an incorrect one. As a result, we obtained an *Inconsistency Dataset* containing 126 inconsistent constraints and 90 consistent constraints. In a sense, our dataset can be considered as potential inconsistencies that may realistically occur during development.

4.1.2 Subjects. Table 1 lists 7 popular libraries that MPChecker evaluated on, first four for dataset construction and last three for assessing MPChecker's ability to detect unknown issues. The selected libraries are of high quality and widely used, with tens of thousands of stars on GitHub. These libraries are substantial third-party libraries, averaging 1,642 classes, 14,467 functions, with an average of 373.6 thousand lines of code, and each function containing an average of 1.3 parameters. Our tool extracts documentation constraints and path constraints from these libraries and uses a fuzzy constraint reasoner to detect inconsistencies.

MPChecker was implemented in Python. All the experiments were performed on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz machine with 252GB of RAM, running Ubuntu 18.04, with Python 3.8.19 and Z3 4.13.0. When evaluating the constraint extraction performance (RQ1), we access the GPT-4 model through OpenAI's API. For result validation, given the absence of established benchmarks in this domain, two volunteer researchers independently reviewed the constraint extraction results from GPT-4, manually assessing each constraint's consistency with the original Python documentation. Any discrepancies were resolved through consensus discussion.

4.2 Results

4.2.1 Accuracy of LLM in extracting constraints from API documentation. We display the result of RQ1 in Table 2. Our experiment shows that our tool correctly identified and extracted 66 out

Table 2. Results of MPChecker on constraint extraction

	Equivalent	Non-Equivalent		Accuracy
	Correct extraction	Incorrect extraction	Missing constraints	
MPChecker w/o few-shot learning	45	20	7	62.5%
MPChecker w/o chain-of-thought	57	7	8	79.2%
MPChecker	66	2	4	91.7%

of 72 constraints contained in the Python documentation collected in our benchmark, achieving an accuracy of 91.7%, which demonstrates that our tool can successfully extract most of the constraints accurately, with few errors or omissions. Next, we look into the remaining failed cases and investigate the reasons for the inaccuracy. We found that out of the 6 incorrect cases, 4 involved missing constraints during the documentation processing. For example, one of the missing constraints is: "(batch_size = auto) → (batch_size = min(200, n_samples))". Although this case involves a constraint between multiple parameters, the format is tricky because one of the parameters is within a function, which may have misled GPT-4 and caused it to miss this constraint during extraction. In the last 2 cases, the constraint information was identified but converted into incorrect logic expressions due to the complex logic or sentence structure.

We further explore MPChecker's abilities by conducting an ablation study. The results show that MPChecker without few-shot learning achieves an accuracy of 62.5%. Most failures occur in the incorrect extraction of constraints. This indicates that including few-shot learning is important for MPChecker to generate accurate constraints. Next, MPChecker without applying chain-of-thought techniques results in an accuracy of 79.2%, with the number of missed constraints accounting for more than half of total non-equivalent cases. This suggests that GPT tends to miss more details in documentation when chain-of-thought is removed. After including chain-of-thought and few-shot learning, MPChecker's performance shows a clear improvement.

Answer to RQ1: MPChecker correctly extracted 66 constraints out of 72 in total, achieving an accuracy of 91.7%. This demonstrates that MPChecker is effective in extracting constraints from Python documentation.

4.2.2 MPChecker's effectiveness in detecting multi-parameter API documentation errors. To measure MPChecker's effectiveness in detecting multi-parameter API documentation errors, we evaluated our tool on the inconsistency dataset. Table 3 shows the results of MPChecker in detecting multi-parameter CDI on the inconsistency dataset.

The Vanilla MPChecker does not include fuzzy words or apply fuzzy constraint satisfaction theory, limiting its ability to handle implicit constraints. Despite these limitations, it achieved an impressive 69% precision by detecting 89 inconsistencies. With the addition of fuzzy words and fuzzy constraints, MPChecker's performance significantly improved, successfully identifying 119 inconsistencies with a precision of 92.8%. This demonstrates that incorporating fuzzy words and fuzzy constraints can expand the range of detectable constraints. However, we observed two false positives. These two false positives occurred because the constraints involved were between parameters and method calls, rather than between the parameters. One example is shown below:

`(shape = None) ∧ (axes ≠ None) → (shape = numpy.take(x.shape,axes,axis=0))`

where, the value for "shape" is the function "take()" from numpy library. Modern software increasingly emphasizes code maintainability and reusability, leading to highly complex function

Table 3. Results of LLM and MPChecker on detecting multi-parameter CDIs

Checker	FP	TP	Precision
LLM	117	7	5.6%
LLM+C	74	52	41.3%
Vanilla MPChecker	2	87	69.0%
Fuzzy MPChecker	2	117	92.8%

LLM: raw documentation and corresponding code; LLM+C: extracted *doc-constraints* from documentation and corresponding code; Vanilla MPChecker: MPChecker without fuzzy words and fuzzy constraint logic; Fuzzy MPChecker: MPChecker with fuzzy words and fuzzy constraint logic.

calls, often involving nested or chained calls. To avoid the high risk of path explosion in symbolic execution, we alternate function calls with symbolic inputs during the preprocessing phase, which leads to misclassification of these two inconsistencies.

The remaining 9 unresolved constraints stem from external function dependencies. While incorporating external function code could resolve these constraints, this approach risks infinite recursive dependencies and path explosion. For us, best practices suggest that constraints should be handled within the documented function itself.

A notable situation arose during one of our issue reporting, even though our issue had been confirmed, we encountered dissatisfaction from one of the developers. He believed we were an automated tool or bot based on AI because of our anonymous status, which diminished his enthusiasm for addressing the issue. With a mass of LLM-based program analysis or inconsistency checkers now available, while they offer insights sometimes, their results often cost more manual verification than traditional tools due to higher uncertainty.

Comparative Study. Therefore, we conducted a comparative experiment between our tool and the approach of using only LLMs as a constraint checker on the same dataset. To align with our experiment settings, we chose GPT-4, one of the leading models, for comparison and evaluated its performance under two settings: 1) LLM: providing the raw documentation and code as input, directly prompting GPT to check for consistency, and 2) LLM+C: This is a two-phase processing. Extracting constraints using the LLM first, and then providing both these constraints and their corresponding code to the LLM for consistency check. In addition, we also require LLM to provide justifications for its answers.

As shown in the Table 3, when raw documentation and the corresponding code were provided as inputs to the LLM, LLM demonstrated significant limitations in detecting multi-parameter CDIs, finding only 7 inconsistencies with a precision of 5.6%. When extracted constraints and code were given as inputs, LLM+C demonstrated heightened awareness of the task and had a higher probability of locating the constraint-related code segments. However, it still struggled to determine inconsistency. Out of 126 inconsistent constraints, 52 were identified correctly, yielding a precision of 41.3%. After a thorough review of LLM's responses, we found that LLM+C gave correct results in many cases but provided unreasonable or even wrong explanations. These results demonstrate that the LLM still has limitations in detecting complicated multi-parameter CDIs, highlighting that our method's design is the key factor in enhancing detection performance rather than any reliance on potential pretraining data leakage.

Answer to RQ2: Large language model (LLM) exhibits limited capability in handling multi-parameter CDIs. Compared to LLM+C with a precision of 41.3%, Fuzzy MPChecker successfully detected 117 out of 126 inconsistent constraints, achieving a 92.8% precision. Notably, Fuzzy MPChecker demonstrated a 23.8% higher precision than Vanilla MPChecker, substantiating the effectiveness of the implementation of fuzzy words and fuzzy constraint logic.

Constraint description of `X` and `Gram` in function `lars_path`

> **X** : None or ndarray of shape (n_samples, n_features)
Input data. Note that **if X is None then the Gram matrix must be specified**, i.e, cannot None or False.

Corresponding code snippet in function `lars_path`

```
def lars_path(...):
    if X is None and Gram is not None:
        raise ValueError("X cannot be None if Gram is not None. Use lars_path_gram to avoid
              passing X and y.")
```

Fixed Constraint description of `X` and `Gram` in function `lars_path`

> **X** : None or ndarray of shape (n_samples, n_features)
Input data. Note that **if X is None, Gram must also be None.**, If only the Gram matrix is available, use lars_path_gram instead.

Fig. 8. Example of the fixed documentation from `Scikit-learn`.

4.2.3 Practical effect of MPChecker. Our tool's effectiveness in detecting unknown multi-parameter inconsistencies was validated by manual review and developer feedback. We reported 14 inconsistencies identified by MPChecker to the library maintenance team, receiving positive engagement and warm responses. Two of them even sparked further discussions about potential issues. This not only affirmed our reports' quality but also reflected the enthusiasm of the open-source community.

For example, an issue confirmed by the `scikit-learn` team originates from the independent function "lars_path", as shown in Figure 8. Apparently, an inconsistency exists between the documentation and code regarding whether "Gram" is None when "X" is None. Therefore, we reported the issue [10] and detailed the documentation sections with inconsistencies alongside its corresponding code snippet. The developer made a bit of archeology, admitted the documentation needed to be updated, and asked if we wanted make a PR to correct this error. Finally, the documentation description was fixed to "If X is None, Gram must also be None".

For most issue reports, we received quick feedback, and 11 inconsistencies were confirmed and improvements were made to documentation or code. Of the remaining three cases, two were reported at the initial phase of our experiments when our understanding of the project architecture was insufficient. The checks for these two constraints are done in other deeper files, but we were unable to verify them accurately at that time. The third inconsistency stemmed from ambiguity in the natural language, which resulted in a different interpretation diverged from the developers' original intent. Furthermore, these reported issues have contributed to four DS/ML repositories (`scikit-learn`, `keras`, `statsmodels`, `dask`), which emphasizes the generalization of our tool.

At the time of writing, 10 out of 11 confirmed inconsistencies have been resolved: 7 through documentation fixes and 3 through updates to both documentation and code. This aligns with intuition: code errors are more likely to cause runtime failures and are thus easier to detect, whereas documentation errors and their potential efficiency impacts are often subtler and harder to identify.

Answer to RQ3: We reported 14 multi-parameter inconsistencies detected by MPChecker to library developers, who have already confirmed 11 inconsistencies by the time of submission (confirmation rate = 78.6%) [1–11]. These results demonstrate that MPChecker can effectively detect unknown API documentation errors. Some of them are even in unseen libraries which highlights its strong generalization capability.

5 Discussions

5.1 Threats to Validity

Internal. There is no established ground truth for multi-parameter code-documentation inconsistencies. To mitigate this, we manually reviewed and verified inconsistencies detected by MPChecker. Given the complexity of multi-parameter constraints, two of our authors spent an additional 10 minutes per inconsistency to verify whether it was a true positive. Moreover, many of the confirmed true positives were further validated by the original library developers, enhancing the credibility of our manual labeling. Additionally, since the GPT-4 model used in our experiments had its last knowledge update in April 2023, and our first issue submissions occurred in February 2024, the risk of data leakage is not a significant concern for our approach.

External. Our tool may not fully generalize to all Python libraries, particularly those outside the data science domain. Although our approach is designed to be broadly applicable, we concentrated on data science libraries due to several practical considerations: (1) they are among the most widely used in the Python ecosystem, (2) they commonly employ the two major docstring formats that MPChecker supports, and (3) they provide well-structured documentation with rich multi-parameter constraints. To enhance the representativeness of our evaluation, we selected high-quality, widely adopted libraries with comprehensive API documentation and accessible source code. A further limitation arises from the unmature Python symbolic execution tools, which may not yet robustly handle all the latest language features. Addressing these challenges will require continued engineering efforts to broaden the applicability of our tool.

5.2 Application Prospects

Our framework's language-agnostic design extends beyond dynamic languages like Python, such as Java, where type information facilitates more comprehensive CDI detection.

Our work represents an effective integration of LLMs and traditional software analysis, with fuzzy constraint logic (FCL) acting as the glue that enables smooth synergy between the two. For example, LLMs have been recently used to infer program specifications from code [38]. In contrast to conventional verification techniques that yield binary outcomes—either `True` or `False`, FCL enables probabilistic evaluation of invariant validity within code contexts. This probabilistic framework will integrate better with LLMs, which may generate close yet incorrect program specifications. In particular, FCL can reduce non-logical errors caused by confusion between similar terms.

6 Related Work

API Documentation Analysis. Numerous empirical studies have revealed the challenges of maintaining high-quality API documentation [12, 13, 15, 20, 26, 29, 35, 42, 57, 60, 65, 77]. These studies indicate that documentation errors are prevalent, even in well-established and widely-used libraries. Additionally, an empirical study from Aghajani et al. [12] shows that linguistic antipatterns in APIs increase the likelihood of developers introducing errors and raising more questions compared to using clean APIs. Saied et al. [57] conducted an observational study focusing on API usage

constraints and their documentation. Zhong and Su [78] proposed a method that combines natural language processing (NLP) with code analysis to identify errors in API documentation, specifically targeting grammatical mistakes (such as spelling errors) and incorrect code references (i.e., names that do not exist in the source code). Lee et al. [32] developed a technique to extract change rules from code revisions and apply them to detect outdated API names in Java documentation, with a particular focus on names of Java classes, methods, and fields.

Another related field is code comments inconsistency [17, 23, 34, 44, 47, 64, 69, 73]. Existing research on code comment analysis predominantly follows two approaches. The traditional method employs program analysis and heuristic rules to detect inconsistencies between comments and the code. Technologies like CUP [37], CUP2 [36], and HebCup [33] exemplify this approach, focusing on automatic just-in-time comment updates when corresponding code changes. The alternative approach leverages NLP techniques, particularly LLMs, to retrieve and extract information from software artifacts.

MPChecker focuses on a distinct problem, specifically on API documentation errors arising from multi-parameter constraints. These issues are more subtle and challenging to detect, particularly within data science libraries built on the dynamic language Python.

LLM-based Program Analysis. A line of research [28, 43, 66, 70, 71, 74, 74] focuses on using LLMs on program analysis. Wadhwa et al. [66] focus on using LLMs to resolve code quality issues in multiple code languages. Several recent researches [28, 70, 71] address applying LLMs on program repairing issues. Nam et al. [43] apply the GPT model to explain code and provide usage details. The existing approaches focus on different purposes compared to MPChecker. Zhang et al. [75, 76] use LLMs to extract constraints from code comments, and apply AST-based program analysis to identify inconsistencies. Rong et al. [54] propose C4RLLaMA, a fine-tuned large language model based on the open-source Code Llama, to detect and correct code comment inconsistencies.

Overall, MPChecker's approach is distinct in two aspects. First, MPChecker specifically focuses on detecting inconsistencies in multi-parameter constraints, which is a missing piece in state-of-the-art works. Next, MPChecker deals with code documentation, which involves longer and more complex text, and is more diverse than most code comments.

7 Conclusion

In this paper, we propose MPChecker, a multi-parameter constraint checker for Python data science libraries. MPChecker utilizes both LLMs and symbolic execution to detect inconsistencies between code and documentation. To mitigate the uncertainty introduced by LLM outputs, we utilize fuzzy constraint logic to accommodate nearly-correct parameter constraints. The experimental results show that MPChecker is effective in identifying multi-parameter API documentation errors. We further reported 14 detected inconsistencies, 11 of which were confirmed by the development team. Our work intuitively explores the multi-parameter constraint inconsistencies between code and documentation, and may inspire more future studies in this field.

8 Data Availability

The source code and dataset are available at https://github.com/ParsifalXu/MPChecker.

9 Acknowledgements

This research is supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23) and the RIE2025 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) (Award I2301E0026), administered by A*STAR, as well as supported by Alibaba Group and NTU Singapore through Alibaba-NTU Global e-Sustainability CorpLab (ANGEL).

References

[1] 2024. Dask issue#11336. https://github.com/dask/dask/issues/11336.
[2] 2024. Keras issue#20141. https://github.com/keras-team/keras/issues/20141.
[3] 2024. Scikit-learn issue#28469. https://github.com/scikit-learn/scikit-learn/issues/28469.
[4] 2024. Scikit-learn issue#28470. https://github.com/scikit-learn/scikit-learn/issues/28470.
[5] 2024. Scikit-learn issue#28473. https://github.com/scikit-learn/scikit-learn/issues/28473.
[6] 2024. Scikit-learn issue#29440. https://github.com/scikit-learn/scikit-learn/issues/29440.
[7] 2024. Scikit-learn issue#29463. https://github.com/scikit-learn/scikit-learn/issues/29463.
[8] 2024. Scikit-learn issue#29464. https://github.com/scikit-learn/scikit-learn/issues/29464.
[9] 2024. Scikit-learn issue#29509. https://github.com/scikit-learn/scikit-learn/issues/29509.
[10] 2024. Scikit-learn issue#30099. https://github.com/scikit-learn/scikit-learn/issues/30099.
[11] 2024. Statsmodels issue#9304. https://github.com/statsmodels/statsmodels/issues/9304.
[12] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2018. A large-scale empirical study on linguistic antipatterns affecting apis. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 25–35.
[13] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 590–601.
[14] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.
[15] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21 (2016), 104–158.
[16] Thomas Ball and Jakub Daniel. 2015. Deconstructing dynamic symbolic execution. In *Dependable Software Systems Engineering*. IOS Press, 26–41.
[17] Arianna Blasi and Alessandra Gorla. 2018. Replicomment: identifying clones in code comments. In *Proceedings of the 26th Conference on Program Comprehension*. 320–323.
[18] Sally C Brailsford, Chris N Potts, and Barbara M Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research* 119, 3 (1999), 557–581.
[19] Alessandro Disney Bruni, Tim Disney, and Cormac Flanagan. 2011. A peer architecture for lightweight symbolic execution. *Universidad de California, Santa Cruz* (2011).
[20] Barthélémy Dagenais and Martin P Robillard. 2010. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 127–136.
[21] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*.
[22] Google. 2024. *AI for every developer.* https://ai.google.dev
[23] Andrew Habib and Michael Pradel. 2018. Is this class thread-safe? inferring documentation using graph-based learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 41–52.
[24] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2024. LM-Infinite: Zero-Shot Extreme Length Generalization for Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 3991–4008.
[25] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 2 (1994), 175–204.
[26] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When not to comment: Questions and tradeoffs with API documentation for C++ projects. In *Proceedings of the 40th International Conference on Software Engineering*. 643–653.
[27] Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. 2024. Llm maybe longlm: Self-extend llm context window without tuning. *arXiv preprint arXiv:2401.01325* (2024).
[28] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs *(ESEC/FSE 2023)*. Association for Computing Machinery.
[29] Hong Jin Kang and David Lo. 2021. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2761–2783.
[30] Ken Kennedy. 1978. Use-definition chains with applications. *Computer Languages* 3, 3 (1978), 163–179.
[31] Bart Kosko and Satoru Isaka. 1993. Fuzzy logic. *Scientific American* 269, 1 (1993), 76–81.

[32] Seonah Lee, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. 2019. Automatic detection and update suggestion for outdated API names in documentation. *IEEE Transactions on Software Engineering* 47, 4 (2019), 653–675.

[33] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Automated comment update: How far are we?. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 36–46.

[34] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. 2014. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 785–790.

[35] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating concept based API element comparison using a knowledge graph. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 834–845.

[36] 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.

[37] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International conference on automated software engineering*. 585–597.

[38] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*.

[39] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. 2006. Soft constraints. In *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 281–328.

[40] Meta. 2024. *Introducing Llama 3*. https://www.llama.com

[41] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837* (2022).

[42] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17 (2012), 703–737.

[43] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, 13 pages.

[44] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. 2019. A framework for writing trigger-action todo comments in executable format. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 385–396.

[45] OpenAI. 2024. *OpenAI Models*. https://platform.openai.com/docs/models/o1

[46] OpenAI. 2024. *What are tokens and how to count them?* https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

[47] 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.

[48] PyExSMT. 2024. Python Symbolic Execution. https://github.com/FedericoAureliano/PyExSMT

[49] PyExZ3. 2024. Python Exploration with Z3. https://github.com/thomasjball/PyExZ3

[50] pySMT. 2024. A library for SMT formulae manipulation and solving. https://github.com/pysmt/pysmt

[51] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology (TIST)* 13, 5 (2022), 1–44.

[52] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 2021. Do comments follow commenting conventions? a case study in java and python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 165–169.

[53] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 112–122.

[54] Guoping Rong, Yongda Yu, Song Liu, Xin Tan, Tianyi Zhang, Haifeng Shen, and Jidong Hu. 2024. Code Comment Inconsistency Detection and Rectification Using a Large Language Model. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 432–443.

[55] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633* (2021).

[56] Zsofi Ruttkay. 1994. Fuzzy constraint satisfaction. In *Proceedings of 1994 IEEE 3rd International Fuzzy Systems Conference*. IEEE, 1263–1268.

[57] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. 2015. An observational study on API usage constraints and their documentation. In *2015 IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 33–42.

[58] Thomas Schiex. 1992. Possibilistic constraint satisfaction problems or "How to handle soft constraints?". In *Uncertainty in Artificial Intelligence*. Elsevier, 268–275.
[59] scikit learn. 2024. Machine learning in Python. https://github.com/scikit-learn/scikit-learn
[60] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *Fundamental Approaches to Software Engineering: 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 14*. Springer, 416–431.
[61] Sphnix. 2024. Example Google Style Python Docstrings. https://www.sphinx-doc.org/en/master/usage/extensions/example_google.html
[62] Sphnix. 2024. Example Numpy Style Python Docstrings. https://www.sphinx-doc.org/en/master/usage/extensions/example_numpy.html
[63] statsmodels. 2024. Statistical modeling and econometrics in Python. https://github.com/statsmodels/statsmodels
[64] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*. Ieee, 83–92.
[65] Gias Uddin and Martin P Robillard. 2015. How API documentation fails. *Ieee software* 32, 4 (2015), 68–75.
[66] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* FSE, Article 36 (2024), 23 pages.
[67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models *(NIPS '22)*. 14 pages.
[68] 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* 35 (2022), 24824–24837.
[69] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.
[70] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831.
[71] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. 2024. CREF: An LLM-Based Conversational Software Repair Framework for Programming Tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, 882–894.
[72] Lotfi A Zadeh. 1965. Fuzzy sets. *Information and Control* (1965).
[73] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 1359–1371.
[74] Yichi Zhang. 2024. Detecting Code Comment Inconsistencies using LLM and Program Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)*. Association for Computing Machinery, 683–685.
[75] Yichi Zhang. 2024. Detecting Code Comment Inconsistencies using LLM and Program Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)*. Association for Computing Machinery, 683–685.
[76] 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 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, 356–366.
[77] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. 2020. An empirical study on API parameter rules. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 899–911.
[78] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 803–816.
[79] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.
[80] Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. 2022. Identifying solidity smart contract api documentation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

Received 2024-10-31; accepted 2025-03-31