Enhancing Software Evolution Reliability through Neurosymbolic Reasoning

Xiufeng Xu

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2026

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

Feb. 2026

Date

Signature

Xiufeng Xu

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

Mar. 2026

Date

Signature

Prof. Yi Li

Authorship Attribution Statement

This thesis contains material from two published peer-reviewed conference papers and one completed manuscript under review, all of which I am listed as an author.

Chapter 4 is published as Xiufeng Xu, Chenguang Zhu, and Yi Li. Compsuite: A dataset of java library upgrade incompatibility issues. In International Conference on Automated Software Engineering. 2023. DOI: 10.1109/ASE56229.2023.00127.

The contributions of the co-authors are as follows:

- Prof. Yi Li and Dr. Chenguang Zhu proposed the topic and provided insightful comments on the methodology and experiment design.
- I constructed the dataset, designed and implemented the tool, performed the experiments, analyzed the data and drafted the manuscript.

Chapter 5 is published as Xiufeng Xu, Fuman Xie, Chenguang Zhu, Guangdong Bai, Sarfraz Khurshid, and Yi Li. Identifying Multi-parameter Constraint Errors in Python Data Science Library API Documentation. In Proceedings of the ACM on Software Engineering. 2025. DOI: 10.1145/3728945.

The contributions of the co-authors are as follows:

- Prof. Yi Li and Dr. Chenguang Zhu proposed the topic and revised the manuscript.
- I co-designed the methodology, implemented and evaluated the approach, and drafted the manuscript.
- Fuman Xie contributed to the implementation of the tool, assisted in conducting the evaluation, and refined the manuscript.
- Prof. Sarfraz Khurshid and Prof. Guangdong Bai provided insightful comments on the methodology design.

Chapter 6 is from a completed paper under review.

The contributions of the co-authors are as follows:

- Prof. Yi Li proposed the topic, supported the research, provided guidance to the methodology design, and revised the manuscript.
- I co-designed the methodology, implemented the tool, conducted the experiments, analyzed the data, and drafted the manuscript.
- Dr. Xiuheng Wu contributed to the implementation of the tool and assisted in conducting the experiments.

- Dr. Zejun Zhang provided insightful comments and helped to write the related works section.

Feb. 2026

Signature

Date

Xiufeng Xu

Acknowledgements

I wish to express my greatest gratitude to my advisor.

"If I had one hour to save the world, I would spend 55 minutes defining the problem and only five minutes finding the solution."

—Einstein, Albert

To my dear family

Abstract

My abstracts

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the context of rapid digital transformation and the accelerated advancement of Artificial Intelligence (AI), software systems have evolved from relatively static artifacts into dynamic entities. To remain effective in the face of changing requirements, technological innovations, and emerging security threats, ongoing modification and extension, collectively known as software evolution, have become an inherent and unavoidable characteristic of modern software engineering. Broadly, software evolution is primarily driven by two forces: **external adaptation**, which responds to changes in the third-party ecosystem, and **internal improvement**, which necessitates the synchronous co-evolution of implementation logic (code) and semantic interface (documentation) to satisfy evolving functional and quality requirements.

The deep integration of AI, especially Large Language Models (LLMs), into the software development lifecycle has signaled a significant paradigm shift in software evolution. Empowered by advances in artificial intelligence, intelligent development tools and autonomous agents are increasingly capable of understanding and processing large-scale codebases alongside natural language requirements, while progressively demonstrating higher-level reasoning abilities such as cross-context analysis and complicated problem decomposition. Leveraging these capabilities, AI systems can automate a wide range of labor-intensive and high-complexity tasks, such as code generation, bug fixing, documentation synchronization, and system

maintenance, thereby reshaping critical stages of software evolution and enabling complex software systems to achieve continuous iteration and scalable growth at unprecedented speed. Meanwhile, this AI-driven development paradigm introduces new technical pathways for managing long-term software evolution, propelling software engineering beyond human-centered development model toward a new stage characterized by the coexistence of human–AI collaboration and autonomous intelligence.

Despite this potential, the application of LLMs in software evolution remains fundamentally constrained by their probabilistic generative nature. Although large-scale parameterization facilitates the emergence of complex reasoning capabilities, LLMs generate outputs based on learned statistical patterns rather than deterministic logic, making it a probabilistic mirage that lacks the deterministic guarantees required for reliability. At the same time, modern software is no longer an isolated entity but a node in a highly interconnected dependency network: a project is not only an internal system but also a third-party library for other projects. Allowing AI agents to arbitrarily modify code and documentation is highly risky, may inadvertently introduce new vulnerabilities or disrupt existing functionalities. In this context, even seemingly minor modifications, such as a library upgrade or internal refactoring, can propagate cascading effects that compromise system reliability, security, and maintainability at scale.

Consequently, ensuring the reliability and robustness throughout AI-assisted software evolution can be distilled into three interrelated core challenges:

- **Managing external evolutionary risks.** Reliable external adaptation is predicated on the ability of AI to foresee and mitigate the risks from third-party library upgrades. However, the scarcity of high-quality, real-world datasets that captures intricate patterns of breaking changes caused by third-party library upgrades prevents AI from evaluating and solving external dependency risks.
- **Maintaining internal semantic alignment.** Internal improvement requires the synchronized update of code and documentation. In practice, code and documentation are often updated asynchronously, leading to semantic drift. Since documentation serves as the outward interface for a project's

dependencies, these internal inconsistencies directly amplify external usage risks.

- **Achieving precise evolution execution.** Precise code localization is the prerequisite for all automated evolution tasks. However, existing LLM-based approaches still rely heavily on superficial textual cues rather than deep structural reasoning. Imprecise identification of modification scopes increases downstream overhead, raises the risk of unintended consequences, and ultimately degrades overall evolution quality.

These three challenges correspond respectively to external adaptation, internal alignment, and execution precision, together forming the central bottleneck in AI-enabled software evolution and providing the systematic research motivation and theoretical foundation for this dissertation.

1.2 Contributions

To address these multifaceted challenges, this dissertation advocates for a systematic and heterogeneous set of solutions. For external adaptation, we prioritize empirical grounding by constructing high-quality datasets that capture real-world breaking change patterns, thereby providing a verifiable basis for AI systems to perceive and reason about evolutionary risks. For internal improvement, our strategy is two-pronged. On one hand, we develop a multi-parameter inconsistency detection framework which combines formal engine with LLM to ensure the semantic integrity between code and documentation. On the other hand, we first formalize the *Keyword Shortcut* bias in current code localization research and propose a novel neurosymbolic agent framework that offloads reasoning to a logic engine to transcend the limitations of textual context in repository-level code localization.

Central to these solutions is the paradigm of neurosymbolic AI, which serves as the methodological foundation of this research. By synthesizing the formal rigor of symbolic logic, such as symbolic execution and Datalog, with the adaptive learning power of Large Language Models, we provide a deterministic anchor for AI-driven evolution, ensuring that complex software modifications are not only contextually aware but also logically verifiable. Figure 1.1 illustrates the structure of this thesis,

which is organized into six chapters. The core contributions of this dissertation are summarized as follows:

- **Reproducible real-world incompatibility dataset.** We construct a dataset, CompSuite, including 123 reproducible, real-world client-library pairs that manifest incompatibility issues when upgrading the library. These data points originate from 88 clients and 104 libraries. We created an automated command-line interface for the dataset. With this interface, users are able to programmatically replicate an incompatibility issue from the dataset with a single command. The interface also offers separate commands for each step involved in the reproduction of incompatibility issues.

- **Multi-parameter code–documentation inconsistency detection.** 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. 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. 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`. 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.

- **Formalization of Keyword Shortcut bias and diagnostic benchmark.** We identify and formalize the *Keyword Shortcut* bias in current code localization research. To address this, we introduce KA-LogicQuery, a diagnostic benchmark specifically designed for Keyword-Agnostic Logical Code Localization (KA-LCL). It contains 25 high-quality purely logical queries with precise ground-truth locations, providing a rigorous testing ground for evaluating the structural reasoning capabilities of LLMs and AI agents. We also introduce KA-LogicQuery-Neg, a variant of KA-LogicQuery where queries

are intentionally modified to ensure their ground-truth sets are empty, to evaluate the abstention capability when no valid location meets the query.

- **Neurosymbolic agent framework for repository-level code localization.** We proposed a novel agent-based framework for repo-level code localization that introduces program facts as an intermediate representation to capture both explicit and implicit code relationships. By synthesizing Datalog queries from natural language, DataLoc offloads intricate structural traversal to a high-performance deterministic reasoning engine, significantly enhancing reasoning capabilities and reducing token consumption. We implement our framework as an automated, end-to-end command-line tool. It features an iterative refinement mechanism where the LLM agent progressively generates and adjusts Datalog rules to navigate repositories. Our tool and benchmark are publicly available at: `https://anonymous.4open.science/r/DataLoc-EFF3`. We conduct an extensive evaluation of DataLoc on both KA-LogicQuery and other issue-driven benchmarks. The experimental results demonstrate that DataLoc significantly outperforms state-of-the-art methods in KA-LCL tasks, achieving superior precision and the capacity for verifiable localization. Furthermore, DataLoc maintains competitive performance on standard issue-driven benchmarks, matching SOTA levels while offering higher reliability in handling negative queries through its deterministic logic.

1.3 Organization

The rest of the thesis is organized as follows:

- **Chapter 2** provides essential background and key concepts necessary for understanding the thesis.
- **Chapter 3** presents CompSuite, a dataset of real-world library upgrade incompatibilities, detailing its construction, characteristics, and potential applications.

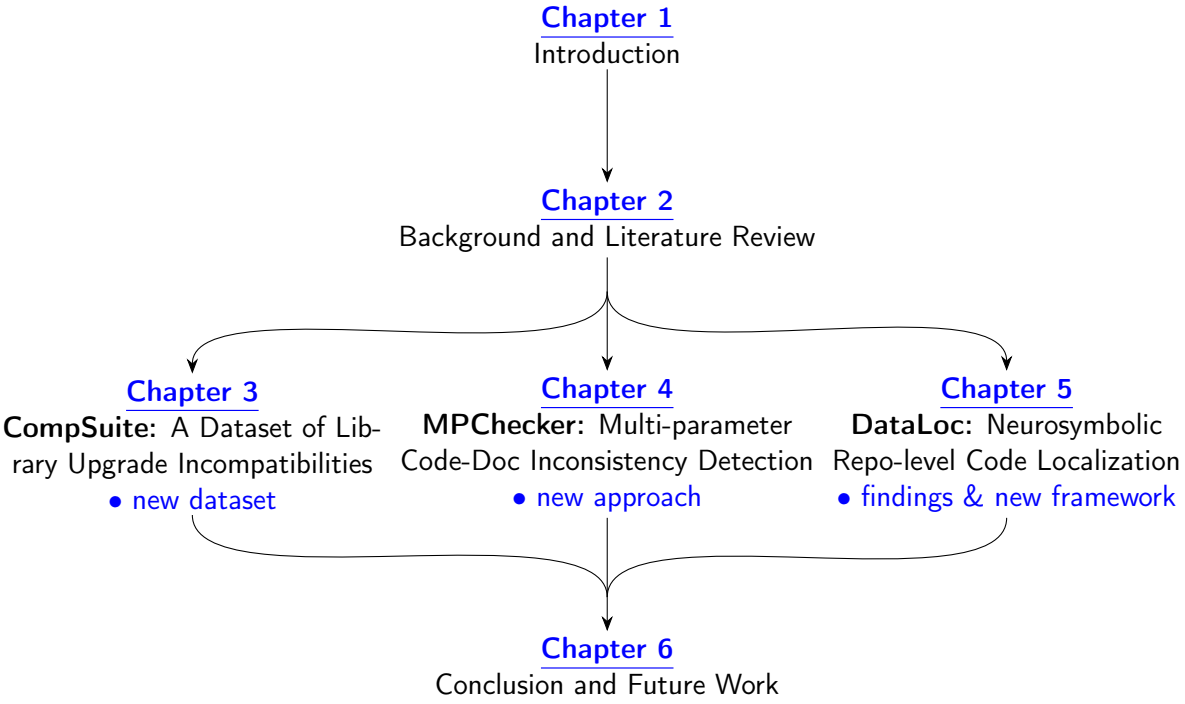


FIGURE 1.1: The Structure of the Thesis

- Chapter 4 introduces MPCHECKER, a multi-parameter code-documentation inconsistency detection tool, describing its design, implementation, and evaluation on real-world libraries.
- Chapter 5 formalizes the keyword shortcut bias in code localization, introduces diagnostic benchmarks, KA-LOGICQUERY and KA-LOGICQUERY-NEG, and presents DATALOC, a neurosymbolic agent framework for repository-level code localization, along with its evaluation results.
- Chapter 6 concludes the thesis by summarizing the contributions, discussing the implications of the findings, and outlining directions for future research in AI-enabled software evolution.

Chapter 2

Background

This chapter presents the essential background and terminology used in subsequent chapters.

2.1 Neuro-Symbolic AI

Symbolic AI focuses on formalized, declarative representations of knowledge through logic and rules, offering high interpretability but struggling with unstructured data. Conversely, Sub-symbolic AI, exemplified by modern deep learning and generative models, learns implicit representations directly from data autonomously. While these methods excel at pattern recognition and scalability, they often lack the explicit logical structures required for rigorous reasoning, transparency, and formal verification [1].

Neuro-symbolic AI draws inspiration from the System 1 and System 2 framework proposed by Kahneman et al. [2]. System 1 is fast, intuitive, and parallel, akin to the capabilities of deep learning, while System 2 is slow, deliberate, and sequential, resembling symbolic reasoning. Neuro-symbolic AI seeks to integrate these two approaches to create systems that leverage the strengths of both and ultimately produce a superior hybrid AI model possessing reasoning abilities. In the context of software engineering, neuro-symbolic AI can be applied to tasks such as program analysis, bug detection, and code localization by leveraging both data-driven learning and rule-based reasoning to achieve more accurate and explainable results.

2.2 Symbolic Execution and SMT Solvers

Symbolic execution is a formal program analysis technique that explores multiple execution paths by treating program inputs as symbolic constants rather than concrete values. Unlike concrete execution, which evaluates a single control-flow path per input, a single symbolic execution represents a class of program behaviors sharing the same execution path. During this process, the execution state is maintained as a functional mapping from program variables to symbolic expressions. For every conditional branch encountered, the analysis identifies the logical conditions necessary to follow different path, accumulating these conditions into a path constraint (π).

The feasibility of an execution path is determined by Satisfiability Modulo Theories (SMT) solvers like Z3 and CVC5. An SMT solver decides whether there exists a concrete assignment of values to the symbolic constants that satisfies the path constraint π. If the constraint is Satisfiable (SAT), the solver provides a model (a concrete test case); if it is Unsatisfiable (UNSAT), the path is unreachable and can be pruned.This mechanism highlights a key trade-off: while symbolic execution is generally less exhaustive than abstract interpretation (which often over-approximates to ensure termination), it achieves higher precision. Any error discovered corresponds to a feasible execution path, effectively eliminating false positives that are common in static analysis.

To bridge the gap between idealized symbolic analysis and real-world program complexity, Concolic Execution (a portmanteau of concrete and symbolic) was introduced. It maintains two simultaneous states: a concrete state tracking actual values, and a symbolic state tracking logical expressions along the same path. The execution typically starts with a random concrete input. As the program runs, the engine records the path constraints. To explore unexplored branches, the engine selects a constraint, negates it, and queries the SMT solver for new concrete inputs.

2.3 Fuzzy Logic and Fuzzy Constraint Satisfaction

Unlike traditional Boolean logic, fuzzy logic [3] 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) [4] 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 [5, 6], 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 [7], 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.

2.4 Program Facts and Datalog

Program facts are a structured representation of information extracted from source code for the purpose of automated reasoning and analysis. They encode observable properties of a program, such as the existence of entities (e.g., functions, classes, variables), their attributes (e.g., names, locations, modifiers), and relations between them (e.g., containment, calls, inheritance), in a form suitable for systematic querying and inference.

Program facts are derived mechanically from source code through language-specific frontends, typically by parsing the code and traversing intermediate representations such as abstract syntax trees or control-flow graphs. Each extracted fact captures a single, well-defined aspect of the program, and together they provide a precise,

machine-readable abstraction of the program's structure and behavior. Importantly, program facts are *descriptive*: they record what is present in the program, rather than how analyses should be performed.

2.4.1 Datalog

Datalog is a declarative logic programming language rooted in first-order logic and database theory. A Datalog specification consists of a set of rules that describe how new facts can be derived from existing ones.

Definition 2.1 (Datalog Rule)**.** A Datalog rule has the form:

$$R_0(t_1, \ldots, t_k) \leftarrow R_1(u_1^{(1)}, \ldots, u_{m_1}^{(1)}), \ldots, R_n(u_1^{(n)}, \ldots, u_{m_n}^{(n)}) \tag{2.1}$$

where each R_i is a predicate symbol. The atom on the left-hand side is called the *head*, and the atoms on the right-hand side form the *body*. Each argument t_j or $u_j^{(i)}$ is either a constant or a variable.

The rule is interpreted as follows: for any assignment of variables to constants that makes all body atoms simultaneously true, the corresponding instantiated head atom is also true. Variables thus serve as placeholders that allow a rule to match and relate multiple facts.

Definition 2.2 (Datalog Program)**.** A Datalog program is a finite set of Datalog rules evaluated over a given set of ground facts. Its semantics is defined as the least fixpoint of rule application: rules are repeatedly applied to derive new ground facts until no further facts can be inferred.

Datalog supports recursion and operates under a monotonic, set-based semantics, making it well suited for expressing transitive and structural properties such as reachability, dependency propagation, and hierarchical relations in program analysis.

2.4.2 Program Facts in Datalog

In program analysis, program facts are represented in Datalog as *ground predicate instances*. Each predicate schema corresponds to a specific kind of program entity or relation, while each extracted program fact instantiates that schema with concrete constants derived from the source code.

For example, a predicate describing function definitions may be declared as:

```
1  .decl function_definition(file_path: symbol, function_name: symbol,
   ↪  start_line: number, end_line: number, param_count: number, is_async:
   ↪  symbol, containing_class: symbol)
```

An extracted function definition in the source code gives rise to a ground fact of this predicate, with all arguments bound to concrete values such as file paths, names, and line numbers.

Datalog rules operate over these ground program facts using variables to range over matching predicate instances. The body of a rule specifies patterns over existing program facts, while the head defines a new fact to be derived whenever the body is satisfied under some variable assignment. Through repeated rule application, Datalog derives higher-level program properties—such as reachability, dependency relations, or structural patterns—from the underlying set of extracted program facts.

This representation cleanly separates *fact extraction*, which records concrete observations about the program, from *logical inference*, which declaratively specifies how additional properties are derived.

Chapter 3

Library Upgrade Incompatibility Issues

3.1 Introduction

Modern software systems are becoming increasingly complex due to the need for integrating various components developed by different teams or organizations. These components are often subject to continuous evolution, and as a result, ensuring that new upgrades to third-party libraries do not cause any compatibility issues with the existing software system is a challenging task. The complexity of these systems and the number of dependencies involved make it difficult to anticipate and identify incompatibilities that may arise from updates to external components. Incompatibility issues resulting from upgrades to external components can compromise the reliability of software systems, potentially leading to significant financial losses for the organizations that rely on these systems.

Many techniques have been proposed to address third-party library compatibility issues, including regression testing [8, 9], static analysis [10], dependency conflict detection [11], and client-specific compatibility checking [12, 13]. These techniques address library compatibility issues in different dimensions and have been evaluated with their own isolated datasets.

An excellent dataset has the potential to serve as a valuable reference for future research in this field. However, composing the dataset requires intricate manual

validation, e.g., confirming whether the cause of a test failure is due to runtime exception, assertion violation, or other reasons. Therefore, we propose CompSuite, the first incompatibility issue dataset focusing on library behavioral incompatibility with concrete reproducible test cases. Each test case is isolated and validated, enabling the direct manifestation of the incompatibilities.

CompSuite comprises 123 real-world Java client-library pairs such that upgrading any library results in incompatibility issues for the corresponding client. Every incompatibility issue in CompSuite contains a test case created by developers, allowing for the reproduction of the issue. On top of this dataset, we also developed an automated command-line interface, which streamlines all processes of the reproduction, such as downloading and compiling a projects, running target tests and re-runing the tests after a library upgrade. With this infrastructure, users may reproduce an incompatibility issue programmatically with minimal efforts.

Contribution To summarize, we make the following contributions in this paper:

1. We construct a dataset, CompSuite, including 123 reproducible, real-world client-library pairs that manifest incompatibility issues when upgrading the library. These data points originate from 88 clients and 104 libraries.
2. We created an automated command-line interface for the dataset. With this interface, users are able to programmatically replicate an incompatibility issue from the dataset with a single command. The interface also offers separate commands for each step involved in the reproduction of incompatibility issues.

We envision that CompSuite to be used to evaluate various program analysis techniques, including compatibility checking, module-level regression testing selection, and debugging techniques. More detailed information can be found in Chapter 3.4.

The dataset and tool are available at: `https://github.com/compsuite-team/compsuite`.

3.2 Dataset Creation

In this section, we outline the methodology and process employed to create the CompSuite dataset.

Table 3.1: Details of clients and libraries included in CompSuite.

Client	**#LoC**	**#Star**
retrofit	29.7K	41.5K
apollo	61.3K	28K
druid	441.9K	26.8K
webmagic	17.4K	10.8K
languagetool	171.2K	8.5K
Other 83 clients (mean)	371.6K	1.3K
All clients (mean)	358.7K	2.5K

Library	**#Maven Usage**
org.slf4j:slf4j-api	62.5K
com.google.guava:guava	34.4K
org.scala-lang:scala-library	34K
com.fasterxml.jackson.core:jackson-databind	25.8K
ch.qos.logback:logback-classic	25.5K
Other 99 libraries (mean)	3.2K
All libraries (mean)	4.8K

3.2.1 Subjects Selection

To ensure the representativeness and reproducibility of the CompSuite dataset, we focus on including high-quality and popular client projects and libraries. The selection of client projects was sourced from GitHub [14], a widely recognized online community for hosting open-source codebases. To ensure the inclusion of the most popular projects, we systematically sorted all the available projects in descending order based on their number of stars on GitHub and selected the target clients from the top of the list. The selection of libraries was sourced from Maven Central [15], which hosts 33.5M of Java libraries and their associated binaries, making it a widely used repository of libraries for Java API and library research [16–19]. We include a library in the dataset only if it has more than 100 usages (i.e., clients) on Maven Central. Our selection criteria aimed to ensure the inclusion of popular and widely used client projects and libraries in the dataset, thereby maximizing its relevance and usefulness to the research community.

Among the highly-rated client projects, our selection criteria focused on those that use Maven [20] as their build systems, given its widespread adoption and maturity. Maven provides a standardized approach to managing Java projects and their dependencies, where each library dependency in a Maven client project is represented as an item in a `pom.xml` file, making it easy to identify and edit library versions programmatically. Furthermore, Maven offers built-in functionality for running unit tests and generating test reports, which simplifies the identification and diagnosis of incompatibility issues arising from test executions. Since Maven projects typically rely on Maven Central as their centralized repository for hosting and downloading libraries, the process of obtaining and managing libraries in our dataset is simplified.

Chapter 3.1 presents the top 5 client projects and libraries in the CompSuite dataset, ranked by popularity. For each client project, we provide information on its lines of code (LoC) and the number of stars it has received on GitHub, while for each library, we include its number of usages by other projects from Maven Central.

In total, CompSuite comprises 123 incompatible client-library pairs. These pairs encompass 88 distinct clients and 104 libraries altogether. On average, the affected clients have 2.5K stars on GitHub and 358.7K lines of code, while incompatible

libraries have 4.8K usages on Maven Central. Thus, we believe that the incompatibility issues present in the CompSuite dataset have a significant impact on a large number of codebases and can affect many users of the libraries, either directly or indirectly.

To ensure that all client projects in the dataset are executable and the runs are reproducible, we performed a series of checks on each project. First, we checked out the project to the version (SHA) at the time of the dataset creation, which we refer to as the *base version*. Next, we ran the standard Maven project compilation command to verify if the project compiles successfully. If the project fails to compile, we excluded it from the dataset. Subsequently, we ran the standard Maven test command to execute all the tests in the project, ensuring that all tests pass on the base version. We excluded any project that fails to pass tests at this stage. Finally, we only included the client projects that successfully compile and pass all tests on the base version, thereby ensuring that the dataset is only consist of projects which can be executed and whose executions can be reproduced.

3.2.2 Data Collection

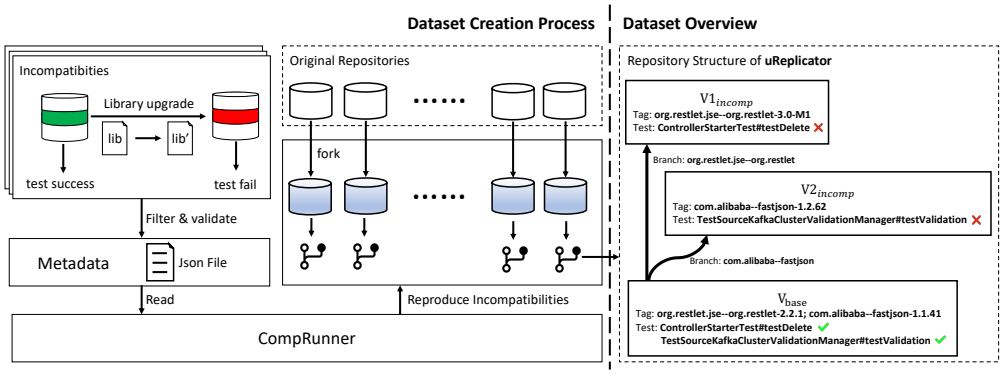


Figure 3.1: The architecture of CompSuite.

We collected the data following the below procedures. Chapter 3.1 visualizes the overall architecture of CompSuite. In the upper left portion of Chapter 3.1, we illustrate the approach taken by CompSuite to identify incompatibilities between a client project and its dependent libraries. Specifically, for each client project on its base version, we upgraded each of its dependent libraries and tested if the upgrade caused any test failures. Our intuition behind this approach is that since all the tests in the client passed on the base version, if upgrading any library causes

```
{
  "id": "i-49",
  "client": "wasabi",
  "sha": "9f2aa5f92e49c3844d787320e2d22e15317aa8e2",
  "url": "https://github.com/intuit/wasabi",
  "lib": "org.apache.httpcomponents:httpclient",
  "old": "4.5.1",
  "new": "4.5.10",
  "test": "DefaultRestEndPointTest#testGetRestEndPointURI",
  "submodule": "modules/export",
  "test_cmd": "mvn
  ↪ org.apache.maven.plugins:maven-surefire-plugin:2.20:test -fn
  ↪ -Drat.ignoreErrors=true -DtrimStackTrace=false
  ↪ -Dtest=DefaultRestEndPointTest#testGetRestEndPointURI"
}
```

FIGURE 3.2: The data schema of COMPSUITE

a test failure, that library upgrade must have introduced incompatibility issues. We refer to the test that flips from passing to failing as an *incompatibility-revealing test*.

To automatically upgrade the libraries and run the tests, we utilized the Maven Versions Plugin [21]. For a given client project, we scanned its dependency list using this plugin to identify all the libraries that had newer versions available on Maven Central. If a library had a newer version, we marked it as upgradable. Next, for each upgradable library, we used the plugin to upgrade it by updating the `pom.xml` file to the most recent version on Maven Central. We then re-executed the test suite of the client. If any tests failed during this run, we marked the client-library pair as having an incompatibility issue and marked the test as an incompatibility-revealing test of this issue. It is crucial to note that we only upgraded one library at a time to isolate failures caused by different libraries. To ensure the accuracy and dependability of the dataset, we carried out a manual verification process for each identified incompatibility issue. In particular, we carefully examined the test failure messages and reports to confirm that they were indeed caused by the upgraded library. For each incompatible client-library pair, we selected a single incompatibility-revealing test to be included in the final dataset. In cases where a client-library pair had multiple incompatibility issues, we chose the one that we deemed most representative and easy to comprehend.

Finally, we persisted the metadata of all the selected incompatibility issues in a collection of `json` files. Chapter 3.2 presents the metadata of an incompatibility

issue in the CompSuite dataset. The data schema includes the ID of the issue, client project name, SHA of the client base version, URL of the client project, library name, versions of the old and new libraries, the name of the incompatibility-revealing test, the submodule containing the incompatibility-revealing test, and the command to run the test. The majority of the information is self-explanatory. However, it is worth noting that the old version of the library is the one utilized at the base version of the client, while the new version is the most recent version found on Maven Central that triggers the incompatibility when upgrading, as described in Chapter 3.2.2.

3.3 Dataset Usage

In this section, we provide instructions on the usage of our dataset.

3.3.1 Exploring an Incompatibility Issue

To ensure the reproducibility of incompatibility issues and to facilitate the demonstration of such issues, we have annotated checkpoints in the version histories of the client projects and provided tags that guide users to explore any incompatibility issues present in the CompSuite dataset.

As illustrated on the right-hand side of Chapter 3.1, our approach to handling incompatible client-library pairs involved creating a fork [22] of the original client project for each identified pair, while preserving all code and version history information. To mark the base version of the project, we utilized the `gittag` [23] command, designating it as V_{base}. Subsequently, we developed a patch to upgrade the library from its old version to its new version, a simple process that can be accomplished with a single line change in the `pom.xml` file for Maven projects. This patch was then applied to the V_{base} version, resulting in a new version that we identified as V_{incomp}. Notably, the only difference between V_{base} and V_{incomp} lies in the library version used: the old (compatible) version is utilized on V_{base} while the new (incompatible) version is utilized on V_{incomp}. For instance, in Chapter 3.1, the client project employs version 2.2.1 of the `org.restlet.jse` – `org.restlet` library on its V_{base} and version 3.0-M1 on its V_{incomp}. In cases where multiple libraries

exhibit incompatibility issues in the client project, we not only create different branches for each library with its name, but also generate a V_{incomp} version tag for each, with accompanying annotations that denote the corresponding library name and version, as depicted in Chapter 3.1.

The V_{incomp} tag for each client-library pair also specifies the specific test that can reveal the incompatibility issue during its run. Following Maven's convention, the test name is formatted as `TestClassName#testMethodName`. By simply copying the text from the tag, users can easily run the incompatibility-revealing test on the V_{incomp} version and observe the incompatibility issue. On the V_{base} version, all tests should pass. This design aims to simplify the usage of CompSuite and make it more accessible and user-friendly.

Using the forked client repositories and version tags provided in the CompSuite dataset, users can easily reproduce any incompatibility issue by checking out to V_{incomp} and running the corresponding incompatibility-revealing test. To compare the behaviors of the client with compatible and incompatible library versions, users can run the incompatibility-revealing test on both V_{base} and V_{incomp} and compare the test outcomes. This allows for a clear understanding of the impact of the library upgrade on the client behaviors.

3.3.2 CompRunner: An Automated Tool for Reproducing Incompatibility Issues

We further developed an automated tool, named CompRunner, which is a part of CompSuite. With CompRunner, users can easily reproduce and investigate any incompatibility issue in a one-click manner by providing the issue ID as input.

We offer an option which enables users to reproduce an incompatibility issue end-to-end with a single command as is shown below. The command outputs and saves all intermediate results and logs for future reference.

```
1 python main.py --incompat i-56
```

When CompRunner runs, it clones the client project from our forked code repository and saves it in the output directory (which is configurable). Then, it checks

out to the base version, compiles the code, and runs the incompatibility-revealing test. Next, it upgrades the library to the new version, reruns the incompatibility-revealing test, and reports any failure information to the user.

We also provide a set of commands that break down the entire cycle of incompatibility exploration into separate steps:

```
1 python main.py --download i-56
2 python main.py --compile i-56
3 python main.py --testold i-56
4 python main.py --testnew i-56
```

We provide several other CompRunner commands for users to inspect different aspects of the incompatibility issues from the CompSuite dataset. A complete list of these commands can be found on CompSuite's website at `https://github.com/compsuite-team/compsuite`.

3.4 Application Scenarios

We anticipate that both researchers and practitioners can benefit from CompSuite to facilitate their investigations and research on errors and test failures induced by library upgrades. CompSuite supports the evaluation of various program analysis techniques, such as software upgrade compatibility checking, debugging, and module-level regression test selection techniques.

As an overview, authors of compatibility checkers and detectors may use CompSuite as a benchmark to evaluate the performance of their techniques against other baseline approaches. Furthermore, authors of debugging techniques can utilize CompSuite as a dataset of compatibility bugs, where each bug corresponds to a test case that verifies the existence or absence of the bug. Finally, authors of module-level regression test selection techniques can use CompSuite to assess the safety of their approaches. A safe module-level RTS technique should select all the corresponding incompatibility-revealing test cases when the library changes.

We detail the three usage scenarios as follows.

- **Compatibility Checkers and Detectors.** The existing techniques for compatibility checking and detection in Java can be categorized into three groups: i) Techniques for detecting API incompatibility that focus on detecting API-breaking changes, such as renaming of code entities and changes in parameter types [24–28]. ii) Techniques for detecting behavioral incompatibility that focus on identifying behavioral differences that cause test failures when a library is upgraded in a client, such as changes in program states [12, 29]. iii) Techniques for detecting dependency conflicts [11, 30, 31], which aim to identify library APIs that exhibit inconsistent semantics between libraries due to class path shading. We believe that developers of techniques in the first two categories can use CompSuite as a benchmark to evaluate their tools' performance, such as precision and recall. They can run their tools on the CompSuite dataset and compare the results with the incompatibility issues present in the dataset. On the other hand, developers of techniques for detecting dependency conflicts can slightly modify CompSuite's dataset by placing both old and new libraries on the class path, running library conflict detection, and checking if the issues can be detected.

- **Module-Level Regression Test Selection.** Regression test selection (RTS) is a technique that aims to reduce the cost of regression testing by selecting a subset of tests that may change the behavior due to code changes on each program version [13, 32–34]. Module-level RTS focuses on selecting the affected client tests when a dependent library is updated [35]. The developers of module-level RTS techniques can evaluate the safety of their tools using CompSuite. For each client-library pair in CompSuite, a module-level RTS tool should select all the corresponding incompatibility-revealing tests when upgrading the library from the old version to the new version.

- **Debugging.** The existing debugging techniques for Java, include symbolic execution [36], delta debugging [37], fault localization [38], etc. These techniques aim to identify the root cause of errors or failures in software. Developers of debugging techniques can use CompSuite as a dataset of compatibility bugs, where each compatibility bug corresponds to a test case that checks the presence or absence of the bug. They can use CompSuite to evaluate their techniques' ability to perform root cause analysis by trying to identify the corresponding library change that caused the compatibility issue.

3.5 related work

To cater to the requirements of various research endeavors, numerous outstanding datasets have been made available to date. Just et al. [39] introduced Defects4J, a database supplies actual bugs, fixed program versions, and corresponding test suites. Bui et al. [40] introduced Vul4J focusing on Java vulnerabilities. Jezek et al. [41] released their dataset of compatibility issues arising from program evolution. There are also many datasets cater for other research domains and ecosystems [42–44].

Distinct from the previously discussed datasets, CompSuite is the first dataset emphasizes the incompatibility issues caused by Java library behavior changes. This type of issues are prevalent and difficult to detect. Additionally, Our developed automated tools also have the capability to assist researchers in swiftly reproducing issues. We believe that a dataset targeting the library upgrade incompatibility issue will contribute to the advancement of the associated technologies.

Chapter 4

Identifying Multi-parameter Constraint Errors in Documentation

4.1 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 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 [45, 46]. 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 [47, 48] 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) [49] 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.

4.2 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

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)
```

FIGURE 4.1: Examples of an explicit constraint from `Statsmodels`.

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.

4.2.0.1 Example 1: Explicit Constraint

The first example comes from `statsmodels` [50], 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. Chapter 4.1 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".

4.2.0.2 Example 2: Implicit Constraint

The second example comes from `scikit-learn` [51], 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. 4.2 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.

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 Chapter 4.3.2.2). 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 Chapter 4.3.3.1).

4.3 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 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 [52–54], API documentation inconsistency can be categorized into two types: incorrectness and incompleteness. Incorrectness refers

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
```

FIGURE 4.2: Examples of implicit constraint from `Scikit-learn`.

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 4.2, 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 5.2, 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

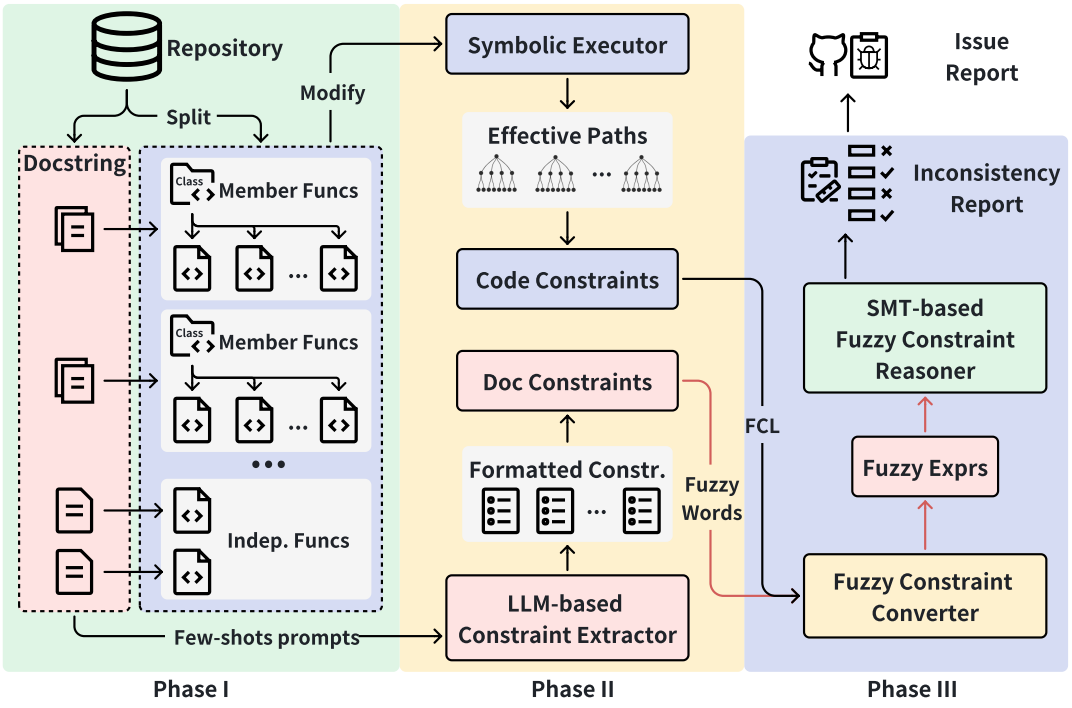


FIGURE 4.3: The architectural overview of MPCHECKER.

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, 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.

4.3.1 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 4.4 respectively display docstring examples of two different styles from Sphinx official website [55, 56]. 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.

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

Numpy Style Docstrings

```
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

```
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
```

FIGURE 4.4: Example of two docstring styles

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.

4.3.2 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.

4.3.2.1 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 [57–61] 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.

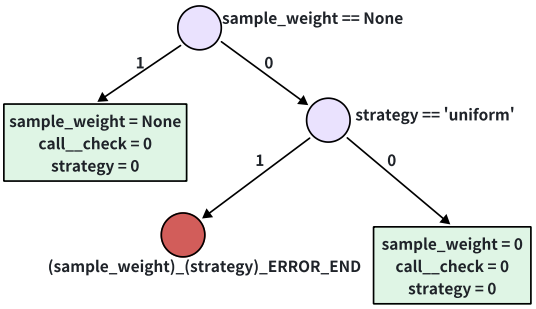
3. Replace the handling of exceptions and warnings that do not affect the path with *return*.

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.

Original source code

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



Modified source code

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

FIGURE 4.5: Extracting constraint from code

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 4.5 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.

4.3.2.2 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 [62], 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 [63] and LLaMA-3 [64] due to its ability to capture details, and it is also well-acquainted with the context of code documentation [65].

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 [66], 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 4.6 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 [67]. We also find that LLMs exhibit

lower performance when dealing with long and complex text inputs as noted in previous research [68, 69]. 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 [70]). We also input the parameter list obtained in Section 4.3.1 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 4.6.

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 [71, 72] 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.

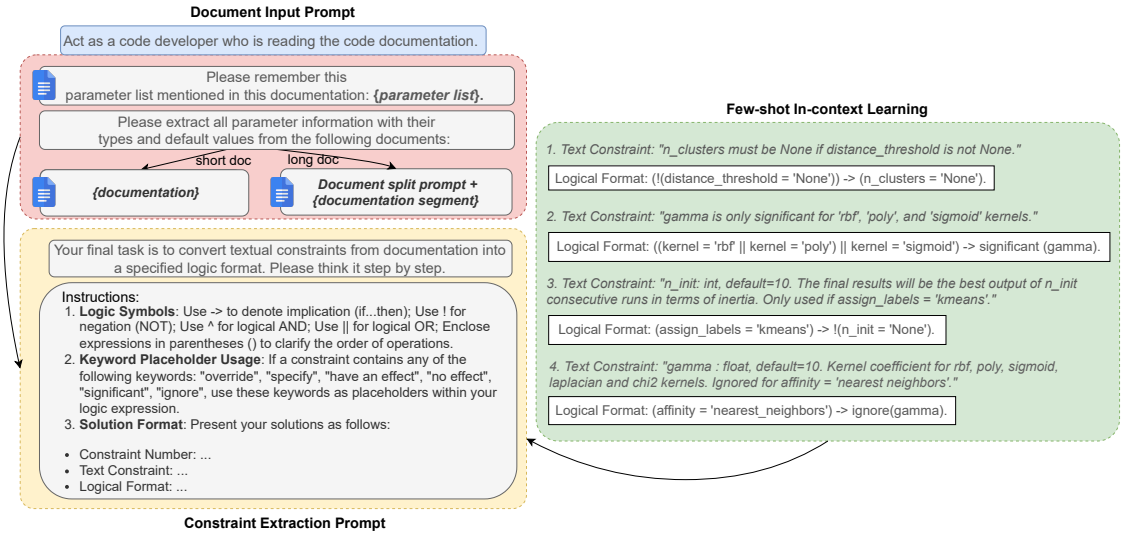


FIGURE 4.6: Prompt structure for constraints extraction

4.3.3 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.

$$\forall p \in P, \neg(c \wedge p) \tag{4.1}$$

- **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.

$$\exists p \in P, \neg(c \Leftrightarrow p) \tag{4.2}$$

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.

4.3.3.1 Fuzzy Words

The example from Chapter 4.2.0.2 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. MPChecker will first trace the target variable's define-use chain (DU-chain) [73, 74] 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 Chapter 4.2.0.2, *gamma* will be ignored not only when ''`affinity=nearest_neighbors`'' but also ignored when ''`affinity=precomputed_nearest_neighbors`'' as well as ''`affinity=precomputed`''.

4.3.3.2 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 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 [3, 7], 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.

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

FIGURE 4.7: Extended Backus-Naur form for multi-parameter constraint.

Chapter 4.7 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 4.1 (Expression Similarity)**.** The similarity between two expressions e_1 and e_2 is defined as,

$$\sigma(e_1, e_2) = \alpha*(1-\frac{LD(p_1, p_2)}{\max(|p_1|, |p_2|)})+\beta*(\frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\|\|\delta_{\bowtie_2}\|})+\alpha*(1-\frac{LD(v_1, v_2)}{\max(|v_1|, |v_2|)}), \quad (4.3)$$

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 Chapter 4.4, where s denotes strings (p or v) and $|s|$ denotes the length of it.

$$\eta(s_1, s_2) = NLD = 1 - \frac{LD(s_1, s_2)}{\max(|s_1|, |s_2|)} \tag{4.4}$$

$$\delta_{\bowtie} = (C, E, G, L, N), \text{where } C, E, G, L, N \in \{0, 1\} \tag{4.5}$$

$$cos\theta(\bowtie_1, \bowtie_2) = \frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\| \|\delta_{\bowtie_2}\|} \tag{4.6}$$

As illustrated in Chapter 4.5, 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 Chapter 4.3.

Definition 4.2 (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.

$$\rho(c, \Phi) = \begin{cases} \arg\max_{e_i \in \Phi} \sigma(e, e_i), & \text{if } c \text{ is an expression } e \\ 1 - \sigma(c', \Phi), & \text{if } c = \neg c' \\ \min\{\sigma(c_1, \Phi), \sigma(c_2, \Phi)\}, & \text{if } c = c_1 \wedge c_2 \\ \max\{\sigma(c_1, \Phi), \sigma(c_2, \Phi)\}, & \text{if } c = c_1 \vee c_2 \end{cases} \tag{4.7}$$

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* [75], 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* [75], when they are combined with a disjunction, the maximum similarity should be used. For negation, the complementary similarity is used.

Definition 4.3 (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:

$$\mu_\Omega(\epsilon) = \rho(\epsilon, \Phi_\Omega) \cdot \epsilon[e \mapsto e_{\Phi_\Omega}] \tag{4.8}$$

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 Chapter 4.1 or Chapter 4.2), 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.

4.3.3.3 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.

4.4 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?

4.4.1 Experiment Setup

4.4.1.1 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 4.4). 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, 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.

Table 4.1: 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	1,535	400.1	1.42	61.8K
pandas	2,211	102	28,880	1,432	620.6	1.14	45.2K
scipy	2,570	142	22,059	1,705	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

4.4.1.2 Subjects

Table 4.1 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.

4.4.2 Results

4.4.2.1 Accuracy of LLM in extracting constraints from API documentation.

We display the result of RQ1 in Table 4.2. Our experiment shows that our tool correctly identified and extracted 66 out 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,

TABLE 4.2: 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%

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.

4.4.2.2 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 4.3 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

TABLE 4.3: Results of LLM and MPCHECKER on detecting multi-parameter CDIs

Checker	FN	TP	Recall
LLM	119	7	5.6%
LLM+C	74	52	41.3%
VANILLA MPCHECKER	39	87	69.0%
FUZZY MPCHECKER	9	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.

limitations, it achieved an impressive 69% recall by detecting 87 inconsistencies. With fuzzy words and fuzzy constraints logic, MPCHECKER's performance significantly improved, successfully identifying 117 inconsistencies with a recall of 92.8% and an accuracy = $\frac{TP+TN}{TP+TN+FP+FN} = \frac{117+88}{216} = 94.9\%$. This demonstrates that incorporating fuzzy words and fuzzy constraints can expand the range of detectable constraints. However, we identified two false positives because the constraints lie between the parameters and method calls, rather than among the parameters. One example is shown below:

$$(\texttt{shape} = \texttt{None}) \land (\texttt{axes} \neq \texttt{None}) \rightarrow (\texttt{shape} = \texttt{numpy.take(x.shape,axes,axis=0)})$$

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 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 inconsistent 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 4.3, 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 recall 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 recall 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 recall of 41.3%, Fuzzy MPChecker successfully detected 117 out of 126 inconsistent constraints, achieving a recall of 92.8%. Notably, Fuzzy MPChecker demonstrated a 23.8% higher recall than Vanilla MPChecker, substantiating the effectiveness of the implementation of fuzzy words and fuzzy constraint logic.

4.4.2.3 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 4.8. Apparently, an inconsistency exists between the documentation and code regarding whether "Gram" is None when "X" is None. Therefore, we reported the issue [76] 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.

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.

FIGURE 4.8: Example of the fixed documentation from `Scikit-learn`.

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%) [76–86]. 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.

4.5 Discussions

4.5.1 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.

4.5.2 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 [87]. 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.

4.6 Related Work

API Documentation Analysis Numerous empirical studies have revealed the challenges of maintaining high-quality API documentation [52, 88–98]. These studies indicate that documentation errors are prevalent, even in well-established and widely-used libraries. Additionally, an empirical study from Aghajani et al. [88] 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. [95] conducted an observational study focusing on API usage constraints and their documentation. Zhong and Su [99] 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. [100] 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 [101–108]. 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 [48], CUP2 [109], and HebCup [110] exemplify this approach, focusing on automatic just-in-time comment updates when corresponding code changes. The alternative approach leverages NLP techniques 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 [111–116, 116] focuses on using LLMs on program analysis. Wadhwa et al. [111] focus on using LLMs to resolve code quality issues in multiple code languages. Several recent researches [112–114] address applying LLMs on program repairing issues. Nam et al. [115] apply the GPT model to explain code and provide usage details. The existing approaches focus on different purposes compared to MPChecker. Zhang et al. [117, 118] use LLMs to extract constraints from code comments, and apply AST-based program

analysis to identify inconsistencies. Rong et al. [119] 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.

Chapter 5

Repo-level Code Localization

5.1 Introduction

The rapid evolution of artificial intelligence has fundamentally reshaped every phase of the Software Development Life Cycle (SDLC), revolutionizing developer-codebase interactions. Through natural language, developers can now gain comprehensive insight into code repositories and perform sophisticated tasks such as code refactoring, feature implementation, and defect repair. At the heart of these capabilities lies the critical primitive of *code localization*. Formally, code localization is the process of precisely identifying relevant source code snippets, ranging from specific methods to complex logic blocks, within a large-scale code repository that aligns with a natural language query. As the community pivots towards autonomous software engineering agents, the efficacy of code localization has emerged as the primary bottleneck. Accurately mapping high-level functional intent to intricate implementation logic is no longer merely a retrieval challenge, but an essential prerequisite for AI systems to reliably navigate and reason about real-world software architectures.

Contemporary state-of-the-art code localization approaches, which can be broadly categorized into three paradigms including embedding-based retrieval, pipeline-guided LLM workflows, and graph-augmented agentic exploration (Details in Section 5.6), have reported impressive results on issue-solving benchmarks such as SWE-bench [120]. However, our investigation reveals a significant but overlooked bias that we term the **Keyword Shortcut**. Recent mainstream benchmarks are

predominantly curated from GitHub issues, which usually contain clear error traces or even verbatim code snippets. Such descriptions are inherently laden with **keywords** (e.g., precise class names or unique identifiers) that act as "cheat sheets", allowing models to locate code snippets via surface-level lexical matching rather than genuine logical reasoning. Our preliminary diagnostic study indicates that once these identifiers are stripped away, the performance of all three approaches suffers a catastrophic decline. This stark contrast uncovers a fundamental deficiency in current systems: a profound struggle with **Keyword-Agnostic Logical Code Localization (KA-LCL)**, where models must navigate codebases without the crutch of explicit naming hints. It is crucial to distinguish KA-LCL from general issue-solving code localization: while the latter is often reducible to a *semantic matching* task between query keywords and code identifiers, KA-LCL represents a higher-order *structural reasoning* challenge. To illustrate this, consider a seemingly straightforward structural logical query: *"Find all functions where: (1) the function has more than 15 parameters, and (2) the function is not an __init__ method"* (Details in Section 5.2). Such an keyword-agnostic logical query poses a significant hurdle for SOTA approaches. While a human developer can easily identify these patterns by traversing the program's structural logic, SOTA solutions frequently fail because they cannot rely on semantic similarity to specific identifiers. Instead, these queries necessitate a deeper understanding of code structures, where existing approaches, deprived of naming cues, prove remarkably brittle.

To systematically evaluate the limits of existing tools, we first introduce KA-LogicQuery, a diagnostic benchmark specifically curated for keyword-agnostic logical code localization. Unlike widely used benchmarks that take issue statements saturated with naming hints, KA-LogicQuery targets scenarios where no key entities serve as anchors. By decomposing code structures, we synthesized a series of purely logical queries that focus on code patterns. Our diagnostic evaluation reveals a precipitous performance degradation in state-of-the-art methods, uncovering a critical "reasoning gap" that current AI-driven localization methods have yet to bridge.

The difficulty of KA-LCL arises from two intertwined technical bottlenecks that current paradigms are ill-equipped to handle. ❶ The absence of lexical anchors leads to an unmanageable search space, significantly exacerbating the *lost-in-the-middle* phenomenon. Modern software systems are typically massive and complex,

and model-based approaches rely heavily on specific identifiers as "pruning signals" to filter out irrelevant modules. Without these *keyword shortcuts*, models are forced to ingest a vast volume of structural context to identify potential candidates. This data deluge overwhelms the limited context window of LLMs, where critical logical patterns become submerged within long input sequences, severely impairing the system's ability to robustly access and utilize relevant structural features. ❷ Existing approaches lack a deterministic reasoning mechanism to navigate the intricate hierarchical dependencies of a repository-level codebase. While pipeline-guided LLM workflows and graph-augmented agentic exploration attempt to address code relationships, they often operate as probabilistic recommendation systems that generate a ranked list of likely candidates, rather than performing rigorous structural deduction. Consequently, the lack of a formal reasoning framework prevents these systems from providing either a deterministic localization or a verifiable explanation, failing to bridge the gap from heuristic-based matching to genuine repository-level logical inference.

Conceptually, code localization can be viewed as a specialized form of code search [121]. However, unlike general information retrieval, programming languages possess formally defined syntax and semantics that allow source code to be precisely parsed and analyzed. This formal nature endows code with an inherent reasonability that extends beyond surface-level text. From a high-level perspective, an effective repository-level localization engine requires a robust intermediate representation (IR) to bridge the semantic gap between natural language intent and implementation logic. Such an IR must effectively encode code entities, their intricate interrelationships, and structural hierarchies, while remaining highly interpretable and actionable for LLM-based agents.

To overcome the aforementioned limitations, we propose DataLoc, a novel agentic framework that synergizes the rule-based reasoning of **Datalog** with the semantic power of LLMs to achieve precise, repository-level code localization. Our framework first employs static analysis to extract a comprehensive set of **program facts** from the source code, constructing a structured IR that captures both elemental properties and relational dependencies. Upon receiving a natural language query, the LLM agent interprets the underlying functional intent and synthesizes a corresponding Datalog query. As a powerful declarative logic programming language,

Datalog is uniquely suited for traversing complex structural patterns that baffle traditional retrieval methods. These queries are then executed by **Soufflé**, a high-performance reasoning engine, which performs rigorous deduction against the pre-extracted facts to infer precise code locations. Crucially, by offloading structural reasoning to a deterministic engine, DataLoc not only significantly reduces token consumption but also empowers the agent to provide definitive negative responses when no matches exist. This avoids the common pitfall of probabilistic systems that hallucinate potential candidates, thereby achieving a paradigm shift from heuristic-based recommendation to verifiable, high-precision localization.

Contributions. Our work aims to integrate Datalog's rule-based inference engine with the advanced large language models. This framework embodies an exploration of the neuro-symbolic paradigm and hope to contribute to open science. In summary, we make the following contributions.

1. We identify and formalize the *Keyword Shortcut* bias in current code localization research. To address this, we introduce KA-LogicQuery, a diagnostic benchmark specifically designed for Keyword-Agnostic Logical Code Localization (KA-LCL). It contains 25 high-quality purely logical queries with precise ground-truth locations, providing a rigorous testing ground for evaluating the structural reasoning capabilities of LLMs and AI agents.

2. We proposed a novel agent-based framework for repo-level code localization that introduces program facts as an intermediate representation to capture both explicit and implicit code relationships. By synthesizing Datalog queries from natural language, DataLoc offloads intricate structural traversal to a high-performance deterministic reasoning engine, significantly enhancing reasoning capabilities and reducing token consumption.

3. We implement our framework as an automated, end-to-end command-line tool. It features an iterative refinement mechanism where the LLM agent progressively generates and adjusts Datalog rules to navigate repositories. Our tool and benchmark are publicly available at: `https://anonymous.4open.science/r/DataLoc-EFF3`.

4. We conduct an extensive evaluation of DataLoc on both KA-LogicQuery and other issue-driven benchmarks. The experimental results demonstrate

Question

```
Find all functions where: (1) the function has more than 15 parameters, and
(2) the function is not an __init__ method.
```

❶ Generated by LLM

Datalog Query

```
% EDB: Facts extracted from source code
.decl function_definition(file_path: symbol, function_name: symbol,
    start_line: number, end_line: number, param_count: number,
    is_async: symbol, containing_class: symbol)

% IDB: Derived analytical facts
.decl LargeFunctions(file_path: symbol, function_name: symbol,
    start_line: number, param_count: number, containing_class: symbol)

% Inference rule for localization
LargeFunctions(file_path, function_name, start_line, param_count,
    containing_class) :-
    function_definition(file_path, function_name, start_line, _,
        param_count, _, containing_class),
    param_count > 15,
    function_name != "__init__".

% Query
.output LargeFunctions
```

❷ Execute by Soufflé

Query Result

```
// file_path                    function_name    start_line   param_count   containing_class
-------------------------------------------------------------------------------------------
astropy/convolution/convolve.py   convolve_fft       442          19         module_level
astropy/io/fits/column.py         _verify_keywords   952          17         Column
```

FIGURE 5.1: A motivating exampele of a logic query

that DATALOC significantly outperforms state-of-the-art methods in KA-LCL tasks, achieving superior precision and the capacity for verifiable localization. Furthermore, DATALOC maintains competitive performance on standard issue-driven benchmarks, matching SOTA levels while offering higher reliability in handling negative queries through its deterministic logic.

5.2 Motivating Example

To illustrate the query process in Datalog, we provide a motivating example. Consider a EA-LCL task that requires identifying functions in a codebase satisfying

specific structural constraints: (1) the function has more than 15 parameters, and (2) the function is not an __init__ method.

Figure 5.1 demonstrates a basic localization process. Given a natural language query, the Large Language Model translates the question's intention into a formal Datalog program. The generated program operates over two types of facts: *Extensional Database (EDB)* facts extracted directly from source code, such as `function_definition` containing metadata about each function's location, parameters, and context; and *Intensional Database (IDB)* facts derived through logical inference, such as `LargeFunctions`. The Datalog query defines an inference rule (lines 8-11) that identifies target functions by matching against `function_definition` facts. Irrelevant attributes are marked with underscore "_" to avoid unnecessary computation, while the rule filters for functions with param_count $>$ 15 and excludes those named "__init__".

When executed by the Soufflé Datalog engine, the query precisely localizes two functions meeting the specified criteria: the `convolve_fft` function with 19 parameters at line 442 in `astropy/convolution/convolve.py`, and the `_verify_keywords` function with 17 parameters at line 952 in `astropy/io/fits/column.py`. This example highlights the key advantages of our approach: the LLM bridges the semantic gap between natural language and formal logic, while Datalog ensures soundness and completeness of results through deductive reasoning over program facts, enabling precise localization without exhaustive manual repository traversal.

5.3 Methodology

In this section, we first formalize the repository-level code localization problem. We then analyze the intrinsic challenges that motivate our hybrid approach, DataLoc, which combines the strengths of large language models with the rigorous logical reasoning of Datalog.

Given a natural language query q (e.g., a bug report or feature request) and a target codebase $\mathcal{C}$, the goal of **repository-level code localization** is to identify a list of relevant code locations $\mathcal{L} = \{l_1, l_2, \ldots, l_k\}$. Effective localization bridges the gap between informal natural language and the rigid execution logic of software. We identify three primary challenges:

- **Challenge 1: Semantic and Lexical Disconnect.** There exists a non-trivial gap between the informal vocabulary of q and the formal identifiers in $\mathcal{C}$. We categorize this disconnect into three progressive layers: (1) *Keyword Absense*, where a query lacks any direct textual anchors present in the code; (2) *Latent Semantic Mapping*, where high-level task descriptions (e.g., `login failure`) lack direct textual overlap with low-level implementation entities (e.g., `AuthManager` or `ValidateToken`); (3) *Lexical Divergence*, where developers use synonyms or abbreviations (e.g., `fqn` for `fullyQualifiedName`) that elude exact match search.

- **Challenge 2: Non-local Structural Dependencies.** Relevant code is often not directly mentioned in queries but connected through dependencies or calls. In modern software, even a single feature may be implemented across multiple files. For example, identifying *password validation* logic may require traversing complex call chains and data flows scattered across authentication modules and database layers. Existing pipeline-based tools rely on local directory traversal and fail to capture these deep, cross-file relational dependencies.

- **Challenge 3: Complex Logical Pattern Constraints.** Questions may contain logical pattern requirements that candidate code must satisfy. Certain localization tasks are defined by structural patterns rather than keywords. For example, *"identifying the conditional statement that raises three distinct error types in different branches"* requires satisfying specific logical constraints defined by the language's syntax and semantics. Such patterns are difficult to locate by simply retrieving class or function information; it requires a more comprehensive understanding of the codebase and reasoning ability.

To address these challenges, we propose DataLoc, a synergy between LLMs and Datalog. Our intuition is that LLMs excel at resolving Challenge 1 by translating vague natural language intents into structured requirements, while Datalog provides the relational and reasoning power to solve Challenge 2 and 3 by performing exhaustive, sound traversal over codebase. In our framework, we represent the codebase $\mathcal{C}$ as a set of pre-extracted program facts $\mathcal{F}$, where $\mathcal{F}$ captures both the source entities and the structural relations (e.g., call graphs, inheritance). Each location $l_i \in \mathcal{L}$ corresponds to a specific level of granularity, such as a file, module, or function, that is essential for resolving the query q. Figure 5.2 illustrates the overview of the framework of DataLoc. Our framework operates in two stages.

First, an offline program fact extraction stage analyzes the codebase to build a structured knowledge base. Second, an automated agent execution stage leverages these extracted facts to perform code localization by synthesizing high-quality Datalog queries.

5.3.1 Program Facts Extraction

In this step, we will discuss the details of extracting program facts from a given repository. We extract program facts through a modular pipeline that separates source discovery, structural parsing, and fact emission. First, we enumerate source files under the repository root using configurable include/exclude patterns that respect version-control ignores, build artifacts, and generated code. Language frontend then analyzes its files using the most suitable intermediate representation. For Python, we parse source code into an abstract syntax tree and emit facts during traversal. The frontend produces Datalog facts annotated with precise source locations and stable identifiers, enabling traceability and incremental updates.

From these frontends, we emit facts describing program entities (files, modules, functions, classes, variables) and relations (containment, inheritance, import/use, call, and reference edges), together with optional control flow and data flow information when available. This representation directly addresses Challenge 2 (non-local structural dependencies) by making cross-file interactions explicit and queryable [122]. Instead of relying on local directory traversal, localization can now operate over global dependency graphs, enabling the identification of code relevant to a feature even when it is scattered across multiple modules and layers.

Meanwhile, our facts enables expressive logical pattern matching, which helps solving the Challenge 3 (complex logical pattern constraints). Structural requirements, such as the presence of specific exception patterns, or multi-step call sequences, can be formulated as Datalog queries over extracted facts. This allows localization tasks defined by program structure rather than surface keywords to be handled systematically, without requiring the LLM to reason over raw source code.

5.3.2 Agentic Workflow

This section elaborates on our agent-based workflow for automated repository-level code localization, which builds upon the program facts constructed offline.

Our end-to-end agent is designed to accept natural language queries and return precise code locations. Unlike approaches that provide a fixed top-n list of candidates, our system outputs a dynamic set of potential locations to maximize precision. To mitigate hallucinations and enhance abstention ability, we decouple reasoning from generation. A deterministic engine handles inference while the LLM functions as a coordinator for query analysis and result calibration. The agent is equipped with three basic tools: "`exec_dl`" for executing Datalog programs, and "`get_file_contents`" along with "`get_sources`" for retrieving source code and specific line ranges.

5.3.2.1 Query Analysis and Information Resolution

The workflow begins with performing a preliminary analysis to extract the core technical concepts and structural elements. By identifying program entities like specific file paths and module names, and core structure descriptions, the agent establishes an internal context. This preparatory step provides the necessary predicates and constraints for the subsequent synthesis of Datalog programs.

5.3.2.2 Synthesize-Check-Refine Loop

As shown in the red box in the Figure 5.2, before execution, DataLoc follows a *synthesize-check-refine* loop to mitigate the impact of hallucinations and improve the executability of LLM-generated Datalog programs. It mainly contains two critical, feedback-driven phases (Details in Section 5.3.3 and 5.3.4): (1) *Syntax correction.* Each synthesized program will undergo a parser-gated validation to ensure syntactic well-formedness. Our workflow adopts a *best-effort repair, then fallback* strategy: unambiguous cases are handled via conservative rule-based fixes, revalidated with Soufflé's parser, and all other cases return error feedback to the LLM.restructuring. (2) *Intermediate-rule diagnosis.* We instrument the program to track row counts of intermediate relations, thereby identifying rules that produce

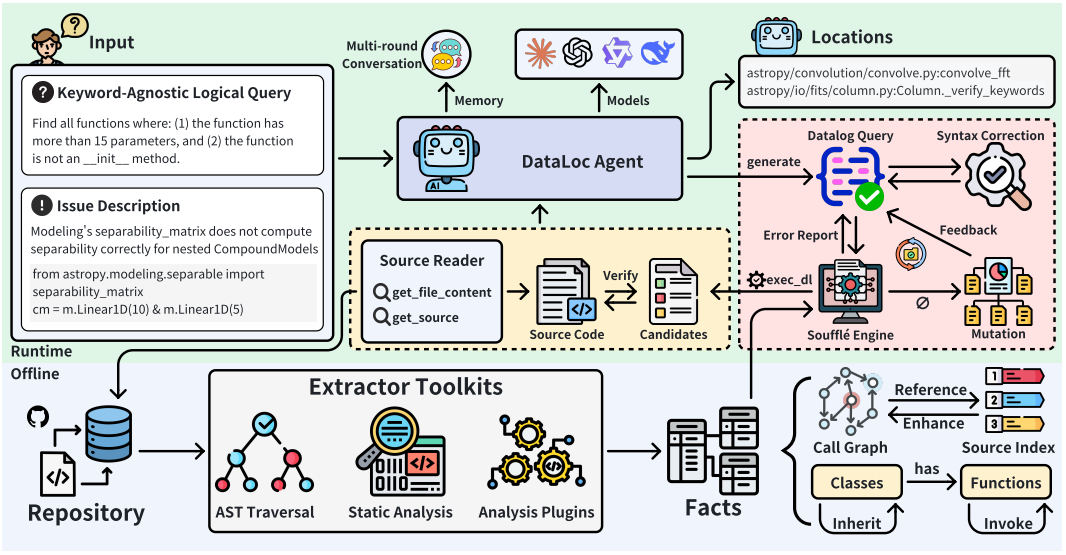


FIGURE 5.2: Overview of DATALOC framework

empty results. Using controlled, mutation-based probing, we differentiate fragile-empty relations caused by over-constraints from stable-empty relations that reflect inherent dataset properties. These feedbacks help the model refine the generated program and ensure its quality before it reaches the inference engine. Validated programs are then executed through `exec_dl` tool to generate a list of candidates. Excessive locations are treated as failures, triggering refinement with stricter constraints to reduce noise and improve precision.

5.3.2.3 Context Retrieval and Final Verification

As depicted in the yellow box in Figure 5.2, once a set of potential code locations has been determined, the agent retrieves the relevant code snippets using `get_sources` or `get_file_contents` and conducts a verification against the original query. These verified locations are then returned in a standardized format (e.g., `FILE_PATH:CLASS.METHOD`). Although the agent may undergo multiple internal reasoning iterations, the user experience is streamlined into a single step: submitting a query and receiving a list of locations. This fully automated closed-loop design ensures both usability and seamless integration into production development environments.

5.3.3 Program Repair for LLM Generated Datalog

In practice, LLM-generated Datalog programs (the dialect used by Soufflé, in our case) often contain syntactic and semantic errors, especially when the model is not fine-tuned for Datalog programming.

Before executing an LLM-generated Datalog program, we enforce a *parser-gated validation* workflow to ensure that only syntactically well-formed programs reach later stages of the pipeline. This design is motivated by the observation that some failures are caused by superficial syntactic issues that can be repaired deterministically, while more complex parse failures often require global restructuring that is better handled by the LLM. Our workflow therefore follows a *best-effort repair, then fallback* strategy: we apply conservative rule-based fixes when the repair is unambiguous, re-check the program using Soufflé's parser, and otherwise return error feedback to the LLM.

We invoke a lightweight parser helper (based on Soufflé's parsing frontend) to validate the generated program. If parsing fails, we first attempt a small set of mechanical rewrite rules targeting high-frequency issues. For example, LLMs frequently introduce naming collisions by using reserved or special identifiers as variable names. E.g., using `count` as a variable name while it is a reserved keyword in Soufflé. Such cases can be fixed locally via deterministic renaming.

However, not all parser errors admit a reliable deterministic repair. In these cases, we do not attempt speculative transformations. Instead, we return the raw parser diagnostics (optionally augmented with concise hints) to the LLM, allowing the model to revise the program directly.

Any program that does not pass the parser check is rejected and never proceeds to semantic validation or execution. Only after the program passes syntactic validation (either initially or after rule-based fixes) do we apply semantic rule checking and subsequent execution. Then we apply a set of semantic correctness checks that encode Soufflé-specific usage rules and common domain conventions observed in LLM-generated programs. These checks target misconceptions that LLMs frequently exhibit when generating Datalog program.

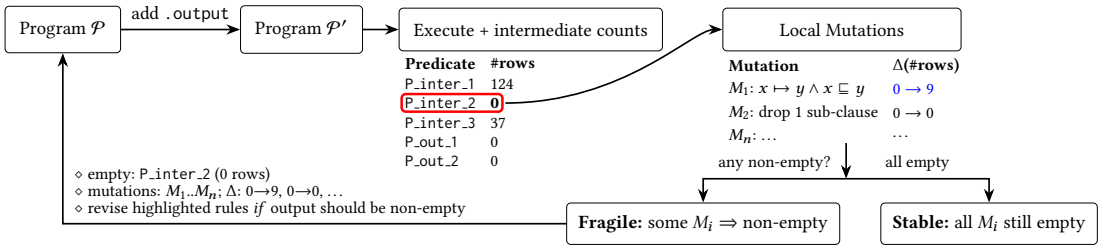


FIGURE 5.3: Intermediate rules mutation and feedback

A representative example is the use of string containment constraints. Soufflé provides a constraint function `contains(sub:symbol,full:symbol)`, which is defined such that the second argument must contain the first (i.e., full includes sub). However, we observe that LLMs frequently invert this positional relationship by producing atoms like `contains(content,"keyword")` instead of the correct `contains("keyword",content)`. Since such inversion conforms to both syntax and type specifications, it leads to silent failure or empty results that are difficult for LLM to self-correct, even with multiple iterations of try and feedback.

Prompt techniques such as few-shot learning cannot effectively eliminate this kind of error. Therefore, we construct a library of semantic rules that check the correct usage of built-in predicates with non-commutative argument semantics and other similar constraints. These repairs are applied only when the transformation is high-confidence and semantics-preserving. When uncertain, the checker records the issue for subsequent feedback to the LLM rather than applying a blind fix, thereby guiding the refinement in subsequent iterations. We evaluate the contribution of this mechanism through an ablation study in 5.4.3.4, demonstrating its effectiveness in improving synthesis quality.

5.3.4 Diagnosing Intermediate Rules via Conservative Mutation Analysis

We introduce an intermediate-rule diagnostic and mutation-based feedback mechanism to improve both the efficiency and effectiveness of LLM-based Datalog synthesis. As illustrated in Chapter 5.3, instead of evaluating a generated program solely by its final output, we instrument execution to collect row counts for intermediate relations and identify rules whose derived relations are empty. In practice, empty

intermediate relations frequently indicate overly restrictive constraints, mismatched join keys, or incorrect predicate usage, and therefore serve as a useful signal for localizing potential errors in synthesized programs.

An empty relation is not inherently incorrect: depending on the user's constraints and the underlying dataset, the semantically correct result may legitimately be the empty set. To avoid forcing spurious revisions or encouraging the model to hallucinate evidence, our approach explicitly distinguishes between fragile and stable emptiness through controlled diagnostic probing.

When detecting an intermediate relation that produces zero rows, we pick several applicable mutations from a small, fixed set of lightweight diagnostic mutations to the corresponding rule and re-execute the program, as shown in Chapter 5.3. These mutations are structure-preserving and intentionally conservative (e.g., relaxing exact string equality to substring or pattern matching, or weakening a single brittle filter), and they are used only for diagnosis, not as candidate replacements for the final query. We then observe whether any mutation yields non-empty results and how row counts change relative to the original execution.

Based on this behavior, we classify empty intermediates into two categories. A relation is considered fragile-empty if at least one diagnostic mutation produces a non-empty result, suggesting that the original rule may be over-constrained or misspecified. In this case, we return targeted feedback identifying the affected relation, the specific mutations applied, and the observed changes in row counts (optionally including a small sample of newly surfaced tuples). Conversely, a relation is considered stable-empty if it remains empty under all tested mutations. In such cases, emptiness may reflect a genuine property of the dataset under the stated constraints rather than a synthesis error. To remain conservative, we do not pressure the model to introduce relaxations; instead, we report that the empty result appears robust and encourage the model to either preserve the current semantics or emit auxiliary diagnostic outputs rather than altering the core query.

Results produced by mutated programs are never used as final answers. Mutations serve only as execution-guided diagnostic probes to help the model decide whether and where revision is warranted. This design balances error localization with semantic caution, enabling targeted repair when evidence exists while avoiding misleading feedback in cases where empty results are likely correct.

5.4 Evaluation

To evaluate the effectiveness and practicality of our approach at repository-level, we design the following research questions:

1. **RQ1:** How effective is DataLoc in keyword-agnostic logical code localization?
2. **RQ2:** How effective is DataLoc for issue-based code localization?
3. **RQ3:** How efficient is DataLoc compared to baselines?
4. **RQ4:** How does each component of DataLoc contribute to its performance?

5.4.1 Experiment Setup

5.4.1.1 Benchmarks.

We evaluate code localization performance on three Python-based benchmarks, covering both complex logical reasoning challenges and industrial issue-resolution tasks.

SWE-bench Lite [120]. A carefully curated and widely recognized subset from the full SWE-bench for more efficient and cost-effective evaluation of autonomous issue-solving capabilities. It consists of real-world GitHub issues with repository metadata and ground-truth patch locations. Following Suresh et al. [123], we retained 274 of 300 original instances where patches modify existing functions or classes. We intentionally excluded instances introducing code corresponding to new functions or import statements to focus the evaluation on code localization within existing structures.

KA-LogicQuery (Ours). To evaluate the capability of localization approaches in keyword-agnostic logical code localization, we constructed KA-LogicQuery, a diagnostic benchmark comprising 25 high-quality logic-intensive queries. As illustrated in Table 5.1, each query is formulated as a composite logical proposition by integrating code features across multiple dimensions. By combining structural granularity (e.g., classes, methods) with behavioral attributes (e.g., control flow,

exception handling) and code metrics (e.g., inheritance depth and branch count), KA-LogicQuery captures complex patterns that demand deep repository understanding.

For each query, we utilize the environment (repository and base commit version) from the first case of SWE-bench Lite as the foundation. For each query, we executed searches using `Cursor` and `GitHub Copilot` in agent mode with multiple latest advanced models like Claude-4.5-Opus and GPT-5.2, and manually validated all returned results to establish the ground-truth locations.

KA-LogicQuery serves as a critical complement to issue-based benchmarks for localization task. In practice, issues are one of the most important channels for error feedback between users and maintainers. To facilitate debugging, those issue descriptions often provide sufficient information and clear keywords as cues to help maintainers better locate faults, such as accurate file paths, function identifiers, or even specific code snippets. Our analysis of SWE-bench Lite instances reveals that over 50% of ground-truth locations are mentioned in the issue descriptions. Such *keyword shortcut* enables models to succeed via simple lexical matching (e.g. grep) or embedding-based retrieval, without requiring genuine understanding and reasoning over the codebase. This undermines the validity of localization performance evaluations. Moreover, LLM-assisted development shifts the codebase interaction toward intent-based question answering, allowing developers to query repositories using natural language. However, for developers unfamiliar with a given repository, they typically cannot use precise identifiers and instead tend to express their search intent through high-level behavioral pattern descriptions or abstract logical structures.

KA-LogicQuery-Neg (Ours). KA-LogicQuery-Neg is a variant of KA-LogicQuery where queries are intentionally modified to ensure their ground-truth sets are empty. Current methods often adopt top-n ranking to maximize recall, but ideal robust localization requires the ability to provide ascertained answers and avoid false positives. KA-LogicQuery-Neg evaluates the abstention capability when no valid location meets the query. Such "refusal" mechanism is a critical metric for ensuring the reliability of autonomous agents in production environments.

TABLE 5.1: Taxonomy of Python Code Dimensions and Representative Elements

Query Dimensions	**Examples / Typical Elements**
Code Structure	Functions, Methods, Classes, Modules, Decorators
Control Flow	Conditional (`if-elif-else`), Iteration (`for`, `while`), Context Management (`with`)
Condition Logic	Comparison (`==`, `>`), Identity (`is`), Membership (`in`), Type Checks (`isinstance`), Logical Operators (`and`, `or`, `not`), Early Exit (`return`, `break`, `continue`)
Data Structure	Built-in Collections (`list`, `dict`, `set`), Primitive Types (`int`, `str`)
Function Signatures	Default Values, Variadic Parameters (`*args`, `**kwargs`), Type Annotations
Exception Handling	Exception Propagation (`try-except-finally`), Exceptions (`TypeError`)
Code Metrics	Nesting Depth, Inheritance Depth, Assertion Count, Branch count

5.4.1.2 Baselines.

To assess DATALOC, we select four state-of-the-art baselines representing three distinct technical paradigms: embedding-based, pipeline-based, and agent-based approaches:

1. **SweRank** [124] (*Embedding-based*): It utilizes a retrieve-and-rerank architecture to identify issue locations. It employs SWERankEmbed (137M/7B parameters) to perform initial retrieval and SWERankLLM (7B/32B parameters) to rerank the results.

2. **Agentless** [125] (*Pipeline-based*): This approaches employs a hierarchical filtering strategy within a procedural workflow. It progressively prunes the search space from the file level down to specific classes or functions, utilizing an LLM to rank and select candidates at each stage.

3. **LocAgent** [126] (*Agent-based*): It constructs a graph-based representation and sparse indexes of the project and enable an autonomous agent to perform iterative, tool-assisted retrieval.

4. **CoSIL** [127] (*Agent-based*): This framework focuses on structural dependency traversal through call graphs to identity implicit locations via iterative exploration. It incorporates pruning to maintain context efficiency and restrict the search to high-relevance execution paths.

5.4.2 Metrics

We evaluate localization performance at three granularity: *file, module, and function*. Let Q denote the set of query instances, G_q the set of ground-truth locations for query $q \in Q$, and $\mathcal{P}_q$ the set of predicted locations inferred by our agent workflow. $\mathbf{1}(\cdot)$ denotes the **indicator function**, which equals 1 if the logical condition holds and 0 otherwise. We adopt the following six metrics:

M1. Accuracy@k (ACC@k): It measures the ability to achieve full coverage, where a success requires all ground-truth locations to be present within the top-k predicted locations. When k equals the length of the prediction set, this metric becomes the **Success Rate (SR)**:

$$Acc@k = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}(G_q \subseteq \mathcal{P}_{q,k}) \tag{5.1}$$

M2. Recall (REC): It represents the proportion of ground-truth locations successfully captured by the predicted set $\mathcal{P}_q$:

$$Rec@k = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap \mathcal{P}_{q,k}|}{|G_q|} \tag{5.2}$$

M3. Precision (PRE): This metric penalizes overprediction by calculating the fraction of predicted locations that are correct:

$$Pre = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap \mathcal{P}_q|}{|\mathcal{P}_q|} \tag{5.3}$$

M4. Average Jaccard Similarity (AJS): It quantifies the overlap between the predicted and ground-truth sets, which penalizes both missing targets and

redundant predictions:

$$AJS = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap \mathcal{P}_q|}{|G_q \cup \mathcal{P}_q|} \tag{5.4}$$

M5. Perfect Location Rate (PLR): The most Stringent metric, measuring the ratio of instances where the predicted set $\mathcal{P}_q$ exactly matches the ground-truth set G_q. A PLR of 1.0 indicates perfect localization without any extraneous noise (i.e., $AJS = 1.0$):

$$PLR = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}(\mathcal{P}_q = G_q) \tag{5.5}$$

M6. Hit Rate (HR): The most lenient metric, measuring the ratio of instances where the predicted set $\mathcal{P}_q$ provides at least one correct location:

$$HR = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}(\mathcal{P}_q \cap G_q \neq \emptyset) \tag{5.6}$$

5.4.2.1 Implementation and environment.

All experiments were conducted on a server equipped with an Intel Xeon Silver 4216 CPU (2.10 GHz) and 62 GB RAM, running Ubuntu 22.04.5 LTS. Our framework was implemented using Python 3.12.11 and the Soufflé 2.4 Datalog engine. To evaluate DataLoc, we accessed `gpt-4o-20240513` via OpenAI's API, `claude-3-5-sonnet-20241022` through AWS Bedrock services, `Deepseek-reasoner` via DeepSeek' API, and `Qwen3-Max` via Alibaba Cloud Service. For baseline comparisons, we instantiated runtime environments according to their respective official specifications and dependency requirements to ensure a fair evaluation.

5.4.3 Results

5.4.3.1 Effectiveness for logic query

As summarized in Table 5.2, DataLoc achieves a decisive lead over all baselines across all metrics and granularities. At the file level, DataLoc reaches a Precision of 65% and a Success Rate of 64%, which is significantly higher than the baseline

methods. Additionally, while all baselines fail completely to achieve perfect location (0% PLR), DataLoc attains a PLR of 44%, indicating the unique advantage of our framework's in capturing code structure and reasoning capabilities in assisting precise localization. To evaluate the generality of our framework, we applied it to Qwen3-Max. The framework achieved strong performance, even surpassing Claude-3.5 on some metrics, suggesting that it generalizes well across different models.

Even with the most lenient metric, Hit Rate (HR), which only requires at least one correct location, baseline performance drops sharply as the granularity shifts from file-level to module-level and function-level. Other metrics even approach zero. It indicates that, when deprived of explicit keywords and forced into deep tracing, they tend to resort to near-random guessing rather than structural reasoning. In contrast, DataLoc demonstrates strong stability, achieving a high HR of around 80%. This robustness proves that DataLoc's success is not a byproduct of a coarse search space but is driven by rigorous, logic-based reasoning.

Baselines typically rely on top-n recommendations to increase the probability of covering relevant locations, but this strategy is inherently a compromise rather than an optimal solution. An effective code localization tool should return results that precisely satisfy the query constraints, since the true number of relevant locations varies across tasks and is not predetermined. To evaluate this capability, we introduce two additional metrics: Average Jaccard Similarity (AJS) and Perfect Localization Rate (PLR). AJS penalizes both false positives and false negatives, while PLR represents the most stringent criterion, requiring the predicted set to exactly match the ground truth (i.e., achieving 100% AJS). For instance, while LocAgent (Claude-3.5) achieves a 44% hit rate at the file level, its AJS is only 1.57%, indicating that true positives are diluted within an inflated candidate set containing substantial noise. By comparison, our approach consistently maintains high AJS scores, reflecting greater precision in returning constraint-satisfying results without extraneous recommendations. This precision is important for industrial deployment, as it reduces the validation overhead for developers or downstream automated agents, improving the efficiency of maintenance workflows.

Our investigation of SWE-bench Lite shows that most issues are highly localized,

involving an average of only 1.15 code changes. This sparsity raises a key question: do existing tools truly pinpoint root causes, or do they merely rely on high-probability guessing within a narrow search space? To examine this, we use KA-LogicQuery-Neg to evaluate whether tools can recognize when no valid location exists. By modifying constraints, we deliberately created a mismatch between the issue description and the codebase, such that the original ground-truth locations are no longer valid. In this setting, the only correct output is a clear "no match found". Unfortunately, all SOTA baselines suffer from a compulsion to guess. They persistently return top-n recommendations even when query prerequisites are not met. This over-eager behavior proves harmful in practice, as confident yet wrong targets mislead downstream agents, wasting computational resources, and risk introducing regression bugs. These findings suggest that the strong performance reported by existing baselines is partially inflated by their recommendation-centric design, which lacks true localization rationale. Notably, DataLoc demonstrates the necessary discernment to abstain when no valid location exists, returning a clear "no match found" response for over 70% of the queries.

TABLE 5.2: Evaluation results on LogicQuery

Methods	**LLM**	**File Level (%)**						**Module Level (%)**						**Function Level (%)**					
		SR	REC	PRE	AJS	PLR	HR	SR	REC	PRE	AJS	PLR	HR	SR	REC	PRE	AJS	PLR	HR
SweRank	Small	4.00	8.00	3.30	2.66	0	20.00	0	0	0	0	0	0	0	0	0	0	0	0
	Large	0	8.13	4.67	3.47	0	20.00	0	6.04	2.92	2.18	0	16.67	0	4.76	2.38	1.87	0	14.29
Agentless	GPT-4o	0	0	0.80	0.50	0	4.00	0	0	0.35	0.28	0	4.17	0	0	0	0	0	0
	Claude-3.5	0	2.00	1.00	0.80	0	4.00	0	2.08	0.60	0.52	0	4.17	0	0	0	0	0	0
LocAgent	GPT-4o	12.00	2.00	1.37	1.12	0	20.00	4.17	0	0.85	0.62	0	12.50	0	0	0	0	0	0
	Claude-3.5	12.00	3.33	1.85	1.78	0	44.00	8.33	2.08	1.20	1.16	0	25.00	4.76	2.38	0.65	0.62	0	19.05
CoSIL	GPT-4o	0	1.00	1.00	0.57	0	4.00	0	0	0	0	0	0	0	0	0	0	0	0
	Claude-3.5	4.00	9.00	4.13	3.24	0	16.00	4.17	6.25	2.22	1.88	0	8.33	4.76	7.14	1.90	1.75	0	9.52
DataLoc	Claude-3.5	**64.00**	**61.00**	62.45	**57.05**	**44.00**	**80.00**	**62.50**	**60.42**	**60.85**	55.12	**41.67**	**79.17**	**61.90**	**62.70**	**63.63**	57.09	**42.86**	**80.95**
	Qwen3-Max	**64.00**	55.00	**65.00**	56.40	**44.00**	68.00	60.00	50.33	60.00	**56.07**	40.00	64.00	56.00	45.53	55.20	**58.73**	40.00	60.00

Answer to RQ1: The results clearly show that DataLoc effectively handles the keyword-agnostic logical code localization challenge, whereas baselines perform poorly when keyword shorts are unavailable. This means these approaches still rely on shallow lexical matching rather than genuine logical reasoning. Furthermore, their performance on the KA-LogicQuery-Neg exposes fundamental weakness in their refusal ability.

5.4.3.2 Effectiveness for general issues

To evaluate the practical utility of DataLoc in real-world software maintenance, we conducted a comparative analysis on the SWE-bench Lite. Existing approaches usually employ top-n strategy, whereas DataLoc operates without a predefined n. Table 5.3 illustrates that DataLoc remains highly competitive.

A critical distinction lies in the recommendation density and target precision. While DataLoc provides instance-specific localization with an average of only 2 candidates per issue, SOTA baselines rely on much broader and often fixed-size candidate sets to improve their accuracy. Specifically, CoSIL and LocAgent typically default to a top-5 recommendation at the function level, while Agentless routinely recommends 5 locations as candidates regardless of the issue's actual complexity. SweRank adopts an even more aggressive strategy, utilizing top-100 rankings.

Consequently, even when Acc@n metrics appear comparable, DataLoc achieves substantially higher precision. By providing a concise and accurate set of entry points, DataLoc minimizes the noise that developers or downstream agents must filter, reducing validation overhead. This precision-centric design also yields substantial gains in resource efficiency (details in Section 5.4.3.3).

Answer to RQ2: DataLoc also demonstrates competitive performance on issue-solving benchmarks. Notably, given that DataLoc produces only 2 candidate locations on average, it offers distinct advantages in recommendation efficiency and in excluding incorrect locations. This precise localization helps reduce the potential overhead of downstream tasks.

TABLE 5.3: Comparison of different methods and models across various localization granularities.

Method	Model	File (%)			Module (%)		Function (%)	
		Acc@1	Acc@3	Acc@5	Acc@5	Acc@10	Acc@5	Acc@10
SweRank	Small	78.10	92.34	94.53	89.05	92.70	79.56	86.13
	Large	83.21	94.89	95.99	90.88	93.43	81.39	88.69
Agentless	gpt-4o	67.50	74.45	74.45	67.15	67.15	55.47	55.47
	claude-3.5	72.63	79.20	79.56	68.98	68.98	58.76	58.76
LocAgent	Qwen2.5-32B(ft)	75.91	90.51	92.70	85.77	87.23	71.90	77.01
	claude-3.5	77.74	91.97	94.16	86.50	87.59	73.36	77.37
DataLoc	gpt-5.1	71.53	77.38	78.47	70.80	72.26	63.14	64.96
	claude-3.5	72.26	80.66	81.02	75.55	75.55	68.98	68.98

5.4.3.3 Efficiency

Figure 5.4 presents a comprehensive comparison using a lollipop chart, where the vertical axis represents average execution time (seconds) and bubble size reflects total token consumption (labeled in thousands). As illustrated, DATALOC (Claude3.5) establishes a new efficiency frontier for KA-LCL tasks, achieving the optimal balance between speed and token consumption with 37 seconds execution time and a 13.5k token consumption. Compared to other approaches, DATALOC exhibits a clear dual advantage in both temporal efficiency and token economy:

Temporal efficiency. DATALOC (Claude-3.5) completes localization in around half minute per task, outperforming all agentic baselines. Even the fastest CoSIL (GPT-4o) remain over 3× slower. The poor performance in Table 5.2 further indicate that, without keyword shortcuts, baseline methods fail to perform meaningful repo-level inference, despite exhibiting imtermediate reasoning steps.

Token economy. Approaches that rely more on agents incur substantially higher token consumption, with LocAgent and Agentless consuming over an order of magnitude more tokens per task than DATALOC. This token explosion reflects a trade-off where tokens are exchanged for intelligence, but this intelligence is currently tie to text. As a result, without keyword shortcuts to guide retrieval, these agents are trapped in multi-round conversation and iterative codebase exploration. In our evaluation, three queries causes LocAgent to enter infinite loops without producing results.

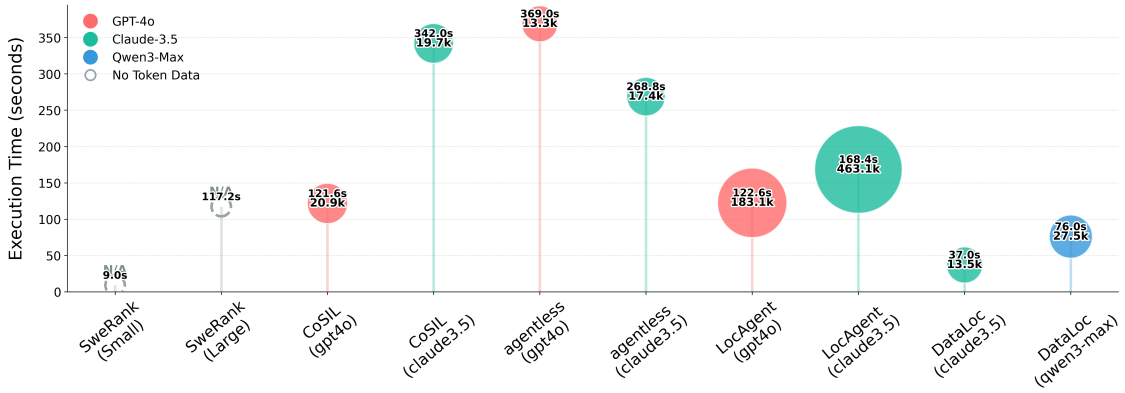


FIGURE 5.4: Average execution time and token consumption of KA-LOGICQUERY

The efficiency of DATALOC stems from its hybrid architecture. In our design, the LLM-based agent is strategically confined to high-level tasks: query analysis, Datalog program synthesis, and final candidate verification. By shifting deep inference to a specialized engine, DATALOC greatly reduces the overhead of redundant multi-round exploration. In addition, errors carry small cost, requiring only the regeneration of a Datalog program. Beyond this low overhead, our parser-gated validation and intermediate-rule feedback actively assist LLM to synthesize high-quality programs.

Answer to RQ3: DATALOC defines the efficiency frontier in the KA-LCL challenge, maintaining an average execution time of 37s and a token consumption of 13.5k. By replacing expensive LLM-based exploration with Datalog-driven inference, DATALOC bypasses the prohibitive costs and execution loops of existing agentic baselines, demonstrating its potential industrial deployment.

5.4.3.4 Ablation Study

We design an ablation study to quantify how two mechanisms detailed in Chapter 5.3.3 and Chapter 5.3.4 improve the quality of LLM-generated Datalog program.

Configurations We evaluate three configurations that progressively enable these mechanisms: **Base** directly executes the LLM-generated Datalog program without any validation or additional feedback, except the output or error messages from Soufflé itself. **VAL** (validation and repair) enables parser-gated validation with

TABLE 5.4: Ablation results under two LLMs. **Base**: no mechanism; **VAL**: validation & repair; **Full**: VAL+intermediate-rule feedback (INT). Rates are reported in %. Iteration and time metrics report mean values (lower is better). Time is reported in seconds.

Model	Config	Rates ↑		Cost (Iter, Time) ↓			Correctness (function-level) ↑					
		ExecSucc	$\neq \varnothing$	First	Iter	Time	SR	PRE	REC	AJS	PLR	HR
Qwen3-Max	Base	73.59	49.41	1.48	12.92	427	20.00	18.20	15.87	26.73	16.00	20.00
	VAL	84.12	75.68	1.28	10.16	104	40.00	38.93	31.87	43.67	32.00	44.00
	Full	84.12	78.52	1.24	10.92	136	56.00	55.20	45.53	58.73	40.00	60.00
Claude-3.5	Base	89.42	67.95	1.16	4.20	38	44.00	38.10	33.87	43.70	24.00	52.00
	VAL	94.67	75.86	1.04	4.28	40	44.00	38.90	31.87	48.36	32.00	52.00
	Full	94.67	80.69	1.08	4.08	37	61.90	62.70	63.63	57.09	42.86	80.95

Cost metrics. $\neq \varnothing$ denotes the fraction of queries that produce non-empty answers. **First** is the mean number of LLM iterations to the first successfully executing program. **Iter** is the average total number of LLM iterations consumed per query. **Time** reports the mean end-to-end wall-clock time per query, including failed attempts and tool feedback.

deterministic syntactic repairs and high-confidence semantic checks. **Full** further enables intermediate-rule mutation and feedback. All configurations share the same LLM, initial prompt, iteration budget, and underlying fact bases.

Metrics We compare (i) execution success rate and non-empty result rate, (ii) mean LLM iterations to first successful execution and (iii) final answer correctness on KA-LOGICQUERY.

5.4 summarizes the ablation results under two different LLMs, comparing the baseline system with progressively enabled mechanisms. The results show that validation and repair (VAL) substantially improves the quality of LLM-generated Datalog for both models, with a markedly stronger effect on Qwen3 Max than on Claude 3.5 Sonnet. This difference is expected given the models' baseline capabilities: without VAL, Qwen3 Max exhibits a significantly lower execution success rate due to its weaker ability to consistently produce syntactically correct Soufflé Datalog, whereas Claude already achieves a relatively high baseline level of syntactic validity.

For Qwen3 Max, enabling VAL leads to a dramatic improvement across nearly all metrics. The non-empty result rate increases from 49% to 75%, indicating that a large fraction of previously failing or unproductive queries were recoverable once parser-gated validation and conservative repairs were applied. At the same time, the average end-to-end execution time drops sharply from 427 seconds to 104 seconds, reflecting a reduction in wasted iterations caused by unrecoverable

parser errors and repeated failed executions. Improvements are also reflected in downstream task quality: function-level correctness metrics show substantial gains, with precision increasing from 18% to 50%, demonstrating that VAL does not merely enable execution but also materially improves the semantic adequacy of the resulting queries. In contrast, the effect of VAL on Claude is more moderate: execution success rate increases by +5 percentage points, and the non-empty result rate increases by +8 percentage points.

Enabling intermediate-rule feedback (INT) in the Full configuration produces a different pattern. The primary role of INT is to guide the LLM toward identifying which specific rule is semantically invalid in the sense of producing no results, and how that rule can be locally revised. As a result, INT may slightly increase iteration count or execution time in some cases due to additional diagnostic executions; however, this overhead consistently translates into higher non-empty rates and improved final answer correctness.

Answer to RQ4: Validation and repair (VAL) substantially improves the robustness of LLM-generated Datalog, with particularly large gains for weaker models by increasing non-empty result rates and reducing execution cost. Intermediate-rule feedback (INT) complements VAL by guiding targeted revisions of logically unproductive rules, occasionally incurring additional diagnostic cost but improving final answer correctness across models.

5.5 Threats to Validity

Internal Validity. The primary internal threats concern baseline implementation and data leakage. We use the official implementations of all baselines with default configurations. For baselines that rely on large language models, we employ the same underlying models (GPT-4o and Claude-3.5-Sonnet) to avoid model-related bias. The KA-LCL queries in KA-LogicQuery and KA-LogicQuery-Neg are constructed based on the decomposed code structure, resulting in novel query instances. Candidate ground truth are generated by state-of-the-art models (GPT-5.2, Claude-4.5-Opus, and Gemini-3-Pro) under advanced AIDE's agent mode and

determined through independent manual verification by two authors. These measures eliminate the risk of data leakage.

External Validity. A main threat to external validity is that our current evaluation focused only on Python. Although the proposed framework is language-agnostic in principle, extending it to additional languages or incorporating more analysis results into program facts remains future work, and addressing this challenge needs further engineering efforts to broaden the applicability.

5.6 Related Work

LLM for Issue Resolution and Question Answering. Large language models are increasingly integrated into software engineering, motivating a wide range of benchmarks for codebase question answering and issue resolution. Existing benchmarks fall into two main categories. Code QA benchmarks [128–130] evaluate models on either code snippets (e.g. CodeQueries [131], CodeQA [132], CoSQA [132]) or repository-level contexts derived from GitHub issues (e.g. CodeRepoQA [133], CoReQA [134], SWE-QA [135]). End-to-end issue resolution benchmarks, such as SWE-Bench [136] and its extensions, assess full issue-solving capabilities [126, 136–147], primarily focus on bug-fix tasks and limited programming languages. However, excessive keywords in these benchmarks provide models with too many shortcuts for code localization by superficial lexical matching. To mitigate this bias, we propose KA-LogicQuery, which removes semantic keywords and only keeps logic structures, and KA-LogicQuery-Neg, an empty-ground-truth variant designed to evaluate abstention ability.

Code Localization. Code localization refers to identifying relevant code locations (e.g., files, modules, or functions) to resolve developers' queries. Recent advancements in this area have taken two complementary directions. Meanwhile, LLM-based retrieval techniques have been proposed to improve code localization performance by leveraging semantic understanding [124–127, 148–153]. Recent research has proposed numerous code localization approaches that can be broadly categorized into three classes: (1) *Embedding-based approaches* (e.g., SWERankEmbed [124], CodeSage [154]) encode code entities and natural language descriptions as embeddings, ranking them based on semantic similarity. While they achieve

high recall by retrieving a broad range of relevant candidate locations, they can only identify code snippets that "look similar" without understanding the logical relationships between them. Furthermore, they suffer from hallucination and noise interference, such as methods with identical names but entirely different functionalities. (2) *Pipeline-based LLM approaches* (e.g., Agentless [125]) follow a structured, multi-stage workflow from files to functions. However, such hierarchical localization design overly depends on initial file-level localization and fails to capture cross-level dependencies, implicitly assuming that developers adhere to good naming conventions. (3) *Agent-based LLM approches* (e.g., LocAgent [126], CoSIL [127], Orca Loca [155], GraphLocator [156]) offer greater flexibility by allowing LLMs to autonomously traverse the repository graph. Nevertheless, they only consider surface-level relevance and exhibit rapid performance degradation without explicit contextual guidance.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis addresses the fundamental tension in modern software engineering: the need for rapid, AI-driven software evolution versus the requirement for deterministic reliability. While Large Language Models (LLMs) have ushered in a paradigm shift toward autonomous intelligence, their probabilistic nature often fails to meet the rigorous demands of interconnected dependency networks. To bridge this gap, this research introduces a neurosymbolic approach that anchors generative intelligence within formal logical frameworks, systematically addressing the three core pillars of evolutionary reliability.

- In **Chapter 3**, we introduce CompSuite, a dataset that includes 123 real-world Java client-library pairs where upgrading the library causes an incompatibility issue in the corresponding client. Each incompatibility issue in CompSuite is associated with a test case authored by the developers, which can be used to reproduce the issue. The dataset also provides a command-line interface that simplifies the execution and validation of each issue. By providing a structured dataset for library upgrade incompatibilities, CompSuite allows AI agents to move beyond anecdotal fixes toward a systematic understanding of how third-party dependency shifts propagate through client systems.

- In **Chapter 4**, we developed 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. This not only ensures internal semantic alignment but also provides a reliable interface contract for external dependencies, mitigating the risks of external usage.

- In **Chapter 5**, we first identify a critical yet overlooked bias: existing real-world issue benchmarks are saturated with keyword references (e.g. file paths, function names), encouraging models to rely on superficial lexical matching rather than genuine structural reasoning. We term this phenomenon the Keyword Shortcut. To address this, we formalize the challenge of Keyword-Agnostic Logical Code Localization (KA-LCL) and introduce KA-LOGICQUERY, a diagnostic benchmark requiring structural reasoning without any naming hints. Our evaluation reveals a catastrophic performance drop of state-of-the-art approaches on KA-LOGICQUERY, exposing their lack of deterministic reasoning capabilities. We propose DATALOC, a novel agentic framework that combines large language models with the rigorous logical reasoning of Datalog for precise localization. DATALOC extracts program facts from the codebase and leverages an LLM to synthesize Datalog programs, with parser-gated validation and mutation-based intermediate-rule diagnostic feedback to ensure correctness and efficiency. The validated programs are executed by a high-performance inference engine, enabling accurate and verifiable localization in a fully automated, closed-loop workflow.

In conclusion, this dissertation demonstrates that the path to reliable software evolution lies not in choosing between human-centered logic and AI-driven automation, but in their synthesis. By embedding neurosymbolic reasoning into the software evolution lifecycle, we provide a theoretical and practical foundation for building autonomous systems that are both as flexible as neural networks and as trustworthy as formal logic.

In conclusion, all of chapters in this dissertation offer a comprehensive exploration of how neurosymbolic approaches can enhance the reliability of software evolution in the age of AI. By systematically addressing the challenges of dependency management, code-documentation consistency, and precise code localization, this research lays the groundwork for a new paradigm of autonomous software maintenance that is both intelligent and dependable.

6.2 Future Research

Bibliography

[1] Brandon C Colelough and William Regli. Neuro-symbolic ai in 2024: A systematic review. *arXiv preprint arXiv:2501.05435*, 2025. 7

[2] Daniel Kahneman. *Thinking, fast and slow*. macmillan, 2011. 7

[3] Bart Kosko and Satoru Isaka. Fuzzy logic. *Scientific American*, 269(1):76–81, 1993. 8, 40

[4] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research*, 119(3):557–581, 1999. 9

[5] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. Soft constraints. In *Foundations of Artificial Intelligence*, volume 2, pages 281–328. Elsevier, 2006. 9

[6] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints?". In *Uncertainty in Artificial Intelligence*, pages 268–275. Elsevier, 1992. 9

[7] Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of 1994 IEEE 3rd International Fuzzy Systems Conference*, pages 1263–1268. IEEE, 1994. 9, 40

[8] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node. js libraries. In *32nd european conference on object-oriented programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 13

[9] Anders Møller and Martin Toldam Torp. Model-based testing of breaking changes in node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 409–419, 2019. 13

[10] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 791–796, 2018. 13

[11] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. Will dependency conflicts affect my program's semantics? *IEEE Transactions on Software Engineering*, 48(7):2295–2316, 2021. 13, 22

[12] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 441–451, 2018. 13, 22

[13] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. A framework for checking regression test selection tools. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 430–441. IEEE, 2019. 13, 22

[14] Github. Github, 2023. `https://github.com/`. 16

[15] Maven. Maven Central Repositories, 2023. `https://mvnrepository.com/`. 16

[16] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. A study on behavioral backward incompatibility bugs in java software libraries. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 127–129. IEEE, 2017. 16

[17] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*, 21:2366–2412, 2016.

[18] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.

[19] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136. IEEE, 2014. 16

[20] Maven. Apache Maven, 2023. `https://maven.apache.org/`. 16

[21] Maven. Maven Versions Plugin, 2023. `https://maven.apache.org/plugins/index.html`. 18

[22] Github. Github Fork, 2023. `https://docs.github.com/en/get-started/quickstart/fork-a-repo`. 19

[23] Github. Github Tag, 2023. `https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/managing-commits/managing-tags`. 19

[24] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018. 22

[25] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 167–177, 2018.

[26] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019.

[27] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 878–888. IEEE, 2019.

[28] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542, 2018. 22

[29] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. Client-specific upgrade compatibility checking via knowledge-guided discovery. *ACM Trans. Softw. Eng. Methodol.*, feb 2023. ISSN 1049-331X. doi: 10.1145/3582569. URL `https://doi.org/10.1145/3582569`. Just Accepted. 22

[30] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 319–330, 2018. 22

[31] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. Could i have a stack trace to examine the dependency conflict issue? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 572–583. IEEE, 2019. 22

[32] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015. 22

[33] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection

in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[34] Lingming Zhang. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, pages 199–209, 2018. 22

[35] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 112–122. IEEE, 2018. 22

[36] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018. 22

[37] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999. 22

[38] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. 22

[39] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014. 23

[40] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 464–468, 2022. 23

[41] Kamil Jezek and Jens Dietrich. Api evolution and compatibility: A data corpus and tool evaluation. *J. Object Technol.*, 16(4):2–1, 2017. 23

[42] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. A dataset for dynamic discovery of semantic changes in version controlled software histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 523–526. IEEE, 2017. 23

[43] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[44] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. Androidcompass: A dataset of android compatibility checks in code repositories. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 535–539. IEEE, 2021. 23

[45] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, ICSE, pages 1199–1210. IEEE, 2019. 25

[46] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 13(5):1–44, 2022. 25

[47] Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 112–122. IEEE, 2017. 26

[48] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 585–597, 2020. 26, 55

[49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 26

[50] statsmodels. Statistical modeling and econometrics in python, 2024. URL `https://github.com/statsmodels/statsmodels`. 28

[51] scikit learn. Machine learning in python, 2024. URL `https://github.com/scikit-learn/scikit-learn`. 29

[52] Gias Uddin and Martin P Robillard. How api documentation fails. *Ieee software*, 32(4):68–75, 2015. 29, 55

[53] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering*, ICSE, pages 27–37. IEEE, 2017.

[54] Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. Identifying solidity smart contract api documentation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 1–13, 2022. 29

[55] Sphnix. Example google style python docstrings, 2024. URL `https://www.sphinx-doc.org/en/master/usage/extensions/example_google.html`. 32

[56] Sphnix. Example numpy style python docstrings, 2024. URL `https://www.sphinx-doc.org/en/master/usage/extensions/example_numpy.html`. 32

[57] PyExSMT. Python symbolic execution, 2024. URL `https://github.com/FedericoAureliano/PyExSMT`. 34

[58] PyExZ3. Python exploration with z3, 2024. URL `https://github.com/thomasjball/PyExZ3`.

[59] pySMT. A library for smt formulae manipulation and solving, 2024. URL `https://github.com/pysmt/pysmt`.

[60] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In *Dependable Software Systems Engineering*, pages 26–41. IOS Press, 2015.

[61] Alessandro Disney Bruni, Tim Disney, and Cormac Flanagan. A peer architecture for lightweight symbolic execution. *Universidad de California, Santa Cruz*, 2011. 34

[62] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 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)*, pages 165–169. IEEE, 2021. 36

[63] Google. Ai for every developer, 2024. URL `https://ai.google.dev`. 36

[64] Meta. Introducing llama 3, 2024. URL `https://www.llama.com`. 36

[65] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024. 36

[66] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Thirty-Sixth Annual Conference on Neural Information Processing Systems*, NeurIPS, 2024. 36

[67] OpenAI. Openai models, 2024. URL `https://platform.openai.com/docs/models/o1`. 36

[68] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 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*, pages 3991–4008, 2024. 37

[69] Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. Llm maybe longlm: Self-extend llm context window without tuning. *arXiv preprint arXiv:2401.01325*, 2024. 37

[70] OpenAI. What are tokens and how to count them?, 2024. URL `https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them`. 37

[71] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022. 37

[72] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633*, 2021. 37

[73] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978. 39

[74] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(2):175–204, 1994. 39

[75] Lotfi A Zadeh. Fuzzy sets. *Information and Control*, 1965. 42, 43

[76] Scikit-learn issue#30099. `https://github.com/scikit-learn/scikit-learn/issues/30099`, 2024. 52, 53

[77] Scikit-learn issue#28469. `https://github.com/scikit-learn/scikit-learn/issues/28469`, 2024.

[78] Scikit-learn issue#28470. `https://github.com/scikit-learn/scikit-learn/issues/28470`, 2024.

[79] Scikit-learn issue#28473. `https://github.com/scikit-learn/scikit-learn/issues/28473`, 2024.

[80] Scikit-learn issue#29440. `https://github.com/scikit-learn/scikit-learn/issues/29440`, 2024.

[81] Scikit-learn issue#29463. `https://github.com/scikit-learn/scikit-learn/issues/29463`, 2024.

[82] Scikit-learn issue#29464. `https://github.com/scikit-learn/scikit-learn/issues/29464`, 2024.

[83] Statsmodels issue#9304. `https://github.com/statsmodels/statsmodels/issues/9304`, 2024.

[84] Scikit-learn issue#29509. `https://github.com/scikit-learn/scikit-learn/issues/29509`, 2024.

[85] Dask issue#11336. `https://github.com/dask/dask/issues/11336`, 2024.

[86] Keras issue#20141. `https://github.com/keras-team/keras/issues/20141`, 2024. 53

[87] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *Proceedings of the 47th International Conference on Software Engineering*, ICSE, April 2025. 54

[88] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*, pages 25–35. IEEE, 2018. 55

[89] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21:104–158, 2016.

[90] Barthélémy Dagenais and Martin P Robillard. 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*, FSE, pages 127–136, 2010.

[91] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE, pages 590–601, 2020.

[92] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. When not to comment: Questions and tradeoffs with api documentation for c++ projects. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE, pages 643–653, 2018.

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

[94] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17:703–737, 2012.

[95] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER)*, pages 33–42. IEEE, 2015. 55

[96] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. An empirical study on evolution of api documentation. In *14th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 14*, pages 416–431. Springer, 2011.

[97] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. An empirical study on api parameter rules. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE, pages 899–911, 2020.

[98] Hong Jin Kang and David Lo. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering*, 48(8):2761–2783, 2021. 55

[99] Hao Zhong and Zhendong Su. Detecting api documentation errors. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 803–816, 2013. 55

[100] Seonah Lee, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. Automatic detection and update suggestion for outdated api names in documentation. *IEEE Transactions on Software Engineering*, 47(4):653–675, 2019. 55

[101] Arianna Blasi and Alessandra Gorla. Replicomment: identifying clones in code comments. In *Proceedings of the 26th Conference on Program Comprehension*, pages 320–323, 2018. 55

[102] Andrew Habib and Michael Pradel. Is this class thread-safe? inferring documentation using graph-based learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE, pages 41–52, 2018.

[103] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE, pages 785–790, 2014.

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

[105] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021.

[106] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension*, ICPC, pages 83–92. IEEE, 2013.

[107] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. Cpc: Automatically classifying and propagating natural language comments via program analysis.

In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE, pages 1359–1371, 2020.

[108] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension*, ICPC, pages 53–64. IEEE, 2019. 55

[109] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, 49(1):1–23, 2021. 55

[110] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. Automated comment update: How far are we? In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, ICPC, pages 36–46. IEEE, 2021. 55

[111] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. Core: Resolving code quality issues using llms. *Proc. ACM Softw. Eng.*, (FSE), 2024. 55

[112] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. ESEC/FSE. Association for Computing Machinery, 2023. 55

[113] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. 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, pages 882–894. ACM, 2024.

[114] Chunqiu Steven Xia and Lingming Zhang. 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, pages 819–831, New York, NY, USA, 2024. Association for Computing Machinery. 55

[115] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE. Association for Computing Machinery, 2024. 55

[116] Yichi Zhang. 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, pages 683–685. Association for Computing Machinery, 2024. 55

[117] Yichi Zhang. 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, pages 683–685. Association for Computing Machinery, 2024. 55

[118] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. 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, pages 356–366. Association for Computing Machinery, 2024. 55

[119] Guoping Rong, Yongda Yu, Song Liu, Xin Tan, Tianyi Zhang, Haifeng Shen, and Jidong Hu. Code comment inconsistency detection and rectification using a large language model. In *2025 IEEE/ACM 47th International Conference on Software Engineering*, ICSE, pages 432–443. IEEE Computer Society, 2024. 56

[120] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? `https://huggingface.co/datasets/SWE-bench/SWE-bench_Lite`, 2024. Accessed: 2025-10-04. 57, 70

[121] Luca Di Grazia and Michael Pradel. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31, 2023. 59

[122] Xiuheng Wu, Chenguang Zhu, and Yi Li. Diffbase: A differential factbase for effective software evolution management. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 503–515, 2021. 64

[123] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code ranking. *arXiv e-prints*, pages arXiv–2412, 2024. 70

[124] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. Swerank: Software issue localization with code ranking. *arXiv preprint arXiv:2505.07849*, 2025. 72, 83

[125] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. 72, 84

[126] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pages 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL `https://aclanthology.org/2025.acl-long.426/`. 72, 83, 84

[127] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Cosil: Software issue localization via llm-driven code repository graph searching. *arXiv preprint arXiv:2503.22424*, 2025. 73, 83, 84

[128] Jan Strich, Florian Schneider, Irina Nikishina, and Chris Biemann. On improving repository-level code qa for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 209–244, 2024. 83

[129] Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. Infibench: Evaluating the question-answering capabilities of code large language models. *Advances in Neural Information Processing Systems*, 37:128668–128698, 2024.

[130] Zehan Li, Jianfei Zhang, Chuantao Yin, Yuanxin Ouyang, and Wenge Rong. Procqa: a large-scale community-based programming question answering dataset for code search. *arXiv preprint arXiv:2403.16702*, 2024. 83

[131] Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and Shirish Shevade. Codequeries: A dataset of semantic queries over code. In *Proceedings of the 17th Innovations in Software Engineering Conference*, pages 1–11, 2024. 83

[132] Chenxiao Liu and Xiaojun Wan. Codeqa: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*, 2021. 83

[133] Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. Coderepoqa: A large-scale benchmark for software engineering question answering. *arXiv preprint arXiv:2412.14764*, 2024. 83

[134] Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and Shuiguang Deng. Coreqa: uncovering potentials of language models in code repository question answering. *arXiv preprint arXiv:2501.03447*, 2025. 83

[135] Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. Swe-qa: Can language models answer repository-level code questions? *arXiv preprint arXiv:2509.14635*, 2025. 83

[136] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*. 83

[137] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

[138] Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. Nocode-bench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130*, 2025.

[139] Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. Crosscodebench: Benchmarking cross-task generalization of source code models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 537–549. IEEE, 2023.

[140] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[141] Yicheng Ouyang, Jun Yang, and Lingming Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 440–452, 2024.

[142] Xinyu Gao, Zhijie Wang, Yang Feng, Lei Ma, Zhenyu Chen, and Baowen Xu. Benchmarking robustness of ai-enabled multi-sensor fusion systems: Challenges and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 871–882, 2023.

[143] Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. Collu-bench: A benchmark for predicting language model hallucinations in code, 2024. URL `https://arxiv.org/abs/2410.09997`.

[144] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024.

[145] Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887, 2024.

[146] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

[147] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL `https://arxiv.org/abs/2504.21798`. 83

[148] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. 83

[149] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[150] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[151] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.

[152] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025.

[153] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *CoRR*, 2025. 83

[154] Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale. *arXiv preprint arXiv:2402.01935*, 2024. 83

[155] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: An llm agent framework for software issue localization, 2025. URL `https://arxiv.org/abs/2502.00350`. 84

[156] Wei Liu, Chao Peng, Pengfei Gao, Aofan Liu, Wei Zhang, Haiyan Zhao, and Zhi Jin. Graphlocator: Graph-guided causal reasoning for issue localization. *arXiv preprint arXiv:2512.22469*, 2025. 84