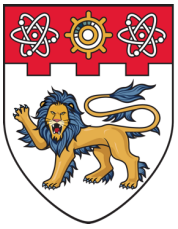
The Greatest Thesis in the World



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

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 Some useful hints

My figure citation: Figure 1.1. (command: fref)

My section citation: Section 1.2. (command: sref)

My Chaptere citation: Chapter 1. (command: cref)

My Paper citation: [1]. (notice back reference to page from bibliograph)

My equation citation: (1.1). (command: eqref), or cite equation by tag: (DOP).

$$F(\theta) = \sum_{i=1}^{m} f_i(\theta) \tag{DOP}$$

$$F(\theta) = \sum_{i=1}^{m} f_i(\theta) \tag{1.1}$$

1.2 Major Contributions

Our main contributions can be stated as follows:

- *First part*: My first contributions, several lines

FIGURE 1.1: An illustration.

- *Second*: Second contributions, several lines
- *Third name*: Third contributions, several lines

1.3 Outline of the Thesis

Chapter 1 introduces ...

Chapter 2 reviews ...

More chapters ....

....

Chapter 2

Literature Review

2.1 Part 1

When you cite a paper [1], the back reference from bibgraph will apper as page number.

You can also cite paper with author name using the command 'citet': such as: Bauschke and Combettes [1].

2.2 Part 2

cite another paper [2].

Lemma 2.1 (My lemma)**.** *A great lemma.*

$$c^2 = a^2 + b^2 \tag{2.1}$$

Theorem 2.2 (My theorem)**.** *A great theorem.*

$$c^2 = a^2 + b^2 \tag{2.2}$$

Proof. The proof is intuitive. □

Corollary 2.3 (My corollary)**.** *A great corollary.*

$$c^2 = a^2 + b^2 \tag{2.3}$$

Proposition 2.1 (My proposition)**.** *A great proposition.*

$$c^2 = a^2 + b^2 \tag{2.4}$$

Example 2.1 (My example)**.** A great example.

$$c^2 = a^2 + b^2 \tag{2.5}$$

Definition 2.1 (My definition)**.** A great definition.

$$c^2 = a^2 + b^2 \tag{2.6}$$

Assumption 2.1 (My assumption)**.** A great assumption.

$$c^2 = a^2 + b^2 \tag{2.7}$$

Remark 2.1 (My remark). A great remark.

$$c^2 = a^2 + b^2 \tag{2.8}$$

Chapter 3

Chapter3 Name

3.1 Section1

See Figure 3.1

FIGURE 3.1: Another illustration.

Let's cite out first table: Table 3.1.

Table	Group 1		Group 2	
	Col 1	Col 2	Col 1	Col 2
Row 1	14.37	**5.76**	2.65	**2.84**
Row 2	5.43	**7.36**	2.22	**2.49**
Row 3	5.54	**5.68**	**4.42**	2.92

TABLE 3.1: My Table.

Chapter 4

MPChecker

Chapter 5

DataLoc

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 [3]. 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 5.2.4). 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 [4]. 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

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 Background

This section introduces the background on program facts and Datalog that forms the symbolic reasoning component of our framework.

5.2.1 Program Facts

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.

5.2.2 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 5.1 (Datalog Rule)**.** A Datalog rule has the form:

$$R_0(t_1, \ldots, t_k) \leftarrow R_1(u_1^{(1)}, \ldots, u_{m_1}^{(1)}), \ldots, R_n(u_1^{(n)}, \ldots, u_{m_n}^{(n)}) \tag{5.1}$$

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 5.2 (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.

5.2.3 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.

5.2.4 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__".

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

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 (nonlocal structural dependencies) by making cross-file interactions explicit and queryable [5].

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.

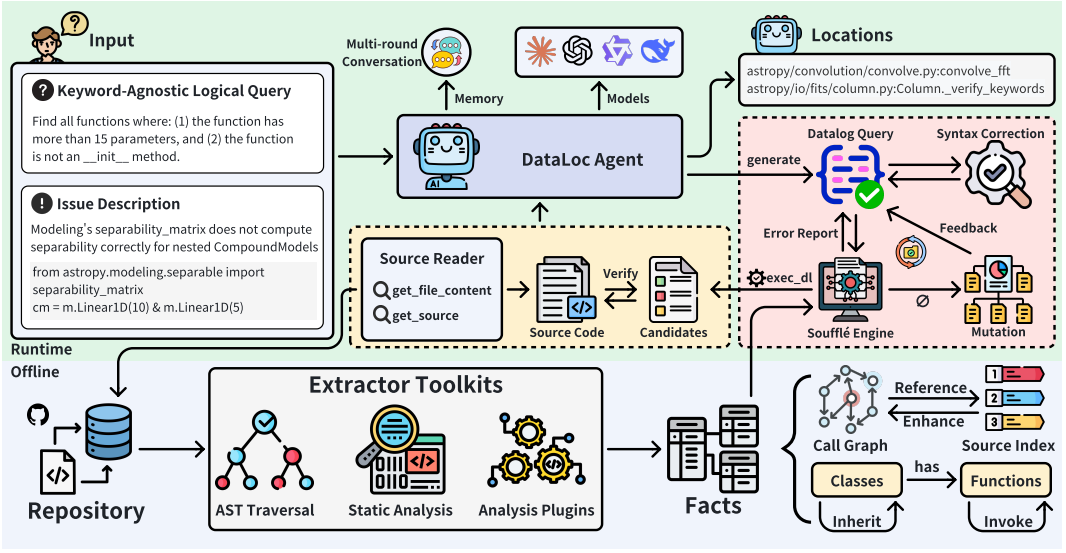


FIGURE 5.2: Overview of DATALOC framework

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 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.

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

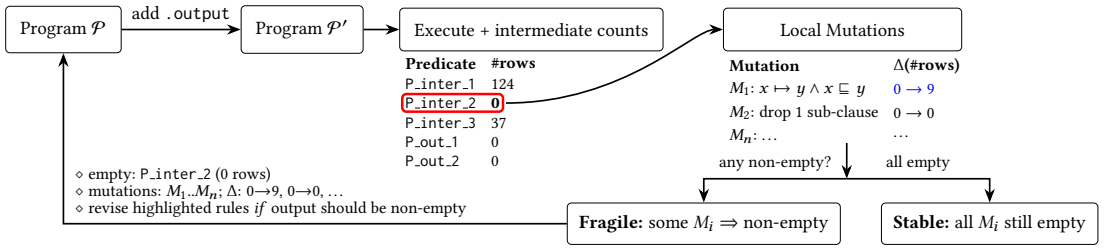


FIGURE 5.3: Intermediate rules mutation and feedback

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 [3]. 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. [6], 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** [7] (*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** [8] (*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** [9] (*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** [10] (*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)**:

$$Acc@k = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}(G_q \subseteq \mathcal{P}_{q,k}) \tag{5.2}$$

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

$$Rec@k = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap \mathcal{P}_{q,k}|}{|G_q|} \tag{5.3}$$

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

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

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

$$AJS = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap \mathcal{P}_q|}{|G_q \cup \mathcal{P}_q|} \tag{5.5}$$

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., $AJS = 1.0$):

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

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:

$$HR = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}(\mathcal{P}_q \cap G_q \neq \emptyset) \tag{5.7}$$

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.

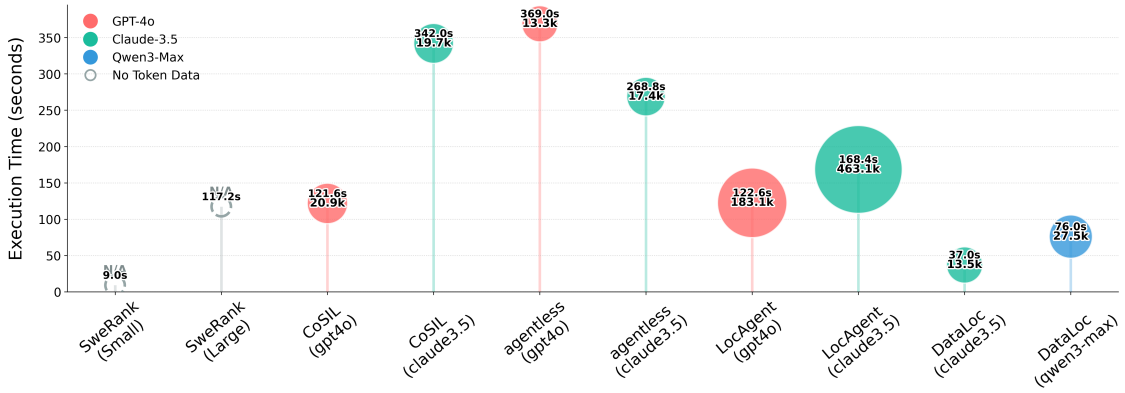


FIGURE 5.4: Average execution time and token consumption of KA-LOGICQUERY

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 [11–13] evaluate models on either code snippets (e.g. CodeQueries [14], CodeQA [15], CoSQA [15]) or repository-level contexts derived from GitHub issues (e.g. CodeRepoQA [16], CoReQA [17], SWE-QA [18]). End-to-end issue resolution benchmarks, such as SWE-Bench [19] and its extensions, assess full issue-solving capabilities [9, 19–30], 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 [7–10, 31–36]. Recent research has proposed numerous code localization approaches that can be broadly categorized into three classes: (1) *Embedding-based approaches* (e.g., SWERankEmbed [7], CodeSage [37]) 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 [8]) 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 [9], CoSIL [10], Orca Loca [38], GraphLocator [39]) 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.

Appendix A

Proofs for Part I or Chapter 3

A.1 Proof of Lemma

$$\psi^{av}(\theta) = \frac{1}{T}\int_0^T [\psi(\theta + \mu(\tau)) + C] \otimes \frac{\mu(\tau)}{a} d\tau$$

A.2 Proof of another Lemma

$$\begin{aligned} \gamma_1(\|x\|) \leq W(t,x) \leq \gamma_2(\|x\|) \\ \frac{\partial W}{\partial t} + \frac{\partial W}{\partial x}\phi(t,x,0) \leq -\gamma_3(\|x\|) \end{aligned} \tag{A.1}$$

List of Author's Awards, Patents, and Publications[1]

Awards

- **Best Paper Awards**, "A Great System," *Nature*.

Patents

- **A Great System**, "A Great System," *Nature*.

Journal Articles

- **My name** and My colleague, "A Great System," *Nature*.

Conference Proceedings

- **My name**, My colleague 1, My colleague 3 and My colleague 3, "Greater System," in *Conference of Vision, 2018*.

[1]The superscript * indicates joint first authors

Bibliography

[1] Heinz H Bauschke and Patrick L Combettes. *Convex analysis and monotone operator theory in Hilbert spaces*. Springer Science & Business Media, 2011. 1, 3

[2] J. B. Rawlings and B. T. Stewart. Coordinating multiple optimization-based controllers: New opportunities and challenges. *Journal of Process Control*, 18: 839–845, 2008. 3

[3] 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. 9, 24

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

[5] 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*, pages 503–515, 2021. 18

[6] 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. 25

[7] 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. 26, 37

[8] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. 27, 38

[9] 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/`. 27, 37, 38

[10] 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. 27, 37, 38

[11] 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. 37

[12] 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.

[13] 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. 37

[14] 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. 37

[15] Chenxiao Liu and Xiaojun Wan. Codeqa: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*, 2021. 37

[16] 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. 37

[17] 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. 37

[18] 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. 37

[19] 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*. 37

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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`.

[27] 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.

[28] 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.

[29] 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.

[30] 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`. 37

[31] 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. 37

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *CoRR*, 2025. 37

[37] 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. 37

[38] 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`. 38

[39] 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. 38