

# PROMETHEUS: Unified Knowledge Graphs for Issue Resolution in Multilingual Codebases

Zimin Chen \*

Sana Lab

Sweden

[zimin@sanalabs.com](mailto:zimin@sanalabs.com)

Yue Pan \*

University College London

United Kingdom

[jack.pan.23@ucl.ac.uk](mailto:jack.pan.23@ucl.ac.uk)

Siyu Lu

KTH Royal Institute of Technology

Sweden

[siyulu@kth.se](mailto:siyulu@kth.se)

Jiayi Xu

University College London

United Kingdom

[wesley.xu.23@ucl.ac.uk](mailto:wesley.xu.23@ucl.ac.uk)

Claire Le Goues

Carnegie Mellon University

United States

[clegoues@cs.cmu.edu](mailto:clegoues@cs.cmu.edu)

Martin Monperrus

KTH Royal Institute of Technology

Sweden

[monperrus@kth.se](mailto:monperrus@kth.se)

He Ye<sup>†</sup>

University College London

United Kingdom

[he.ye@ucl.ac.uk](mailto:he.ye@ucl.ac.uk)

## Abstract

Language model (LM) agents, such as SWE-agent and OpenHands, have made progress toward automated issue resolution. However, existing approaches are often limited to Python-only issues and rely on pre-constructed containers in SWE-bench with reproduced issues, restricting their applicability to real-world and work for multi-language repositories.

We present PROMETHEUS, designed to resolve real-world issues beyond benchmark settings. PROMETHEUS is a multi-agent system that transforms an entire code repository into a unified knowledge graph to guide context retrieval for issue resolution. PROMETHEUS encodes files, abstract syntax trees, and natural language text into a graph of typed nodes and five general edge types to support multiple programming languages. PROMETHEUS uses Neo4j for graph persistence, enabling scalable and structured reasoning over large codebases.

Integrated by the DeepSeek-V3 model, PROMETHEUS resolves 28.67% and 13.7% of issues on SWE-bench Lite and SWE-bench Multilingual, respectively, with an average API cost of \$0.23 and \$0.38 per issue. PROMETHEUS resolves 10 unique issues not addressed by prior work and is the first to demonstrate effectiveness across seven programming languages. Moreover, it shows the ability to resolve real-world GitHub issues in the LangChain and OpenHands repositories. We have open-sourced PROMETHEUS at: <https://github.com/Pantheon-temple/Prometheus>

### ACM Reference Format:

Zimin Chen \*, Yue Pan \*, Siyu Lu, Jiayi Xu, Claire Le Goues, Martin Monperrus, and He Ye<sup>†</sup>. 2025. PROMETHEUS: Unified Knowledge Graphs for Issue Resolution in Multilingual Codebases . In . ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 Introduction

Recent advances in large language models (LLMs) have enabled strong performance across a range of natural language and code-related tasks, including bug fixing [5, 21, 69], code generation

[10, 35, 37, 47], and vulnerability detection [61]. Coding-specific LLMs such as CodeLlama [41], StarCoder [30], JetBrains Junie [72], GPT-4, and Claude Code have shown human-level performance in many function-level tasks. However, most evaluations have focused on short, self-contained problems, and the application of LLMs to repository-scale software engineering tasks remains relatively underexplored.

To address this gap, SWE-bench [23] was introduced as a benchmark for evaluating LLMs on real-world GitHub issue resolution. Code agents, or agentic AI software engineers [40], such as SWE-agent[63] and OpenHands [2], have made progress toward automating software engineering tasks in large codebases with cross-file dependencies and deep context.

Several code agents have been developed to tackle SWE-bench-style issue resolution. Agentless applies an LLM agent framework without repository-level context modeling. Aider [15] leverages file-level diffs and edit suggestions but operates on limited contextual understanding. AutoCodeRover [74] introduces LLM-based repair using dynamic execution traces. RepoGraph [34] proposes a static code graph to support program reasoning but remains scoped to Python. SemAgent [36] adopts a semantics-aware workflow that integrates issue semantics, code semantics, and execution semantics for patch generation.

Despite their success in GitHub issue resolution and strong performance on SWE-bench, prior work faces two key limitations in terms of applicability:

**Problem 1 – Lack of multi-language support:** Prior approaches are tightly coupled to Python, as they focus exclusively on SWE-bench, which consists only of Python repositories. As a result, it remains unclear whether these approaches generalize to multi-language codebases.

**Problem 2 – Lack of issue reproduction:** Prior approaches rely heavily on SWE-bench’s pre-constructed Docker containers, which are specifically built for issue reproduction. In real-world scenarios, agents must autonomously generate build commands and

\*Equal contribution. <sup>†</sup>Corresponding author. Email: [he.ye@ucl.ac.uk](mailto:he.ye@ucl.ac.uk).

reproduce issues without access to predefined environments. Automated project build and issue reproduction are essential steps in the issue resolution process, yet they have been largely overlooked in prior work.

To address these limitations, we propose PROMETHEUS, an approach that constructs a codebase as a unified knowledge graph using general relationships between nodes to support multiple programming languages. PROMETHEUS is a multi-agent system designed for real-world issue resolution beyond benchmark settings. The novelty of PROMETHEUS lies in the inclusion of an Issue Classification Agent and an Issue Reproduction Agent, which take arbitrary GitHub repositories and commit IDs as input, scan open issues, and initiate resolution for those identified as bug-related.

PROMETHEUS can be integrated with any LLM. In our evaluation, it is powered by DeepSeek-V3 to balance cost and effectiveness. We evaluate the performance of PROMETHEUS on two datasets: 300 Python issues from SWE-bench Lite, and 300 issues from SWE-bench Multilingual covering seven programming languages (Java, Rust, C/C++, JavaScript/TypeScript, Ruby, PHP, and Go).

Experimental results show that PROMETHEUS resolves 28.67% of issues in SWE-bench Lite, outperforming several baselines, including AutoCodeRover [74], Agentless [54], and Aider [15], all of which are powered by the more costly GPT-4o. Moreover, PROMETHEUS generates 10 unique correct patches compared with Agentless, AutoCodeRover and RepoGraph [34].

On SWE-bench Multilingual, PROMETHEUS resolves 13.7% of issues and successfully addresses issues across all seven supported languages. PROMETHEUS maintains a low API cost of \$0.23 per issue on SWE-bench Lite and \$0.38 per issue on SWE-bench Multilingual. To our knowledge, PROMETHEUS is the first system evaluated on SWE-bench Multilingual, demonstrating its effectiveness in supporting multiple programming languages.

Additionally, we evaluate PROMETHEUS on real-world eight open issues from LangChain [24] and the OpenHands [2] repositories. Four patches are successfully generated and submitted to the original repositories, thanks to the design of Issue Classification Agent and Issue Reproduction Agent integrated in PROMETHEUS.

In summary, we make the following contributions:

- We propose PROMETHEUS, which constructs the codebase as a unified knowledge graph to support multiple programming languages.
- PROMETHEUS focuses on real-world issue resolution, including automated project build and issue reproduction through customized build reasoning.
- PROMETHEUS resolves 28.67% of issues in SWE-bench Lite, outperforming several prior works and with 10 unique resolved issues.
- PROMETHEUS is the first to evaluate and resolve 13.7% of issues in SWE-bench Multilingual across seven programming languages.
- PROMETHEUS maintains low API cost, averaging \$0.23 per issue on SWE-bench Lite and \$0.38 on SWE-bench Multilingual.
- PROMETHEUS is open-sourced for reproducibility and community sharing.

## 2 Approach

Figure 1 illustrates the architecture of PROMETHEUS, a multi-agent approach designed for automated issue resolution by converting a codebase to a unified knowledge graph.

PROMETHEUS receives two inputs: (1) a GitHub repository URL and (2) a specific commit ID (see Part A). It begins by parsing the project’s directory structure, processing each file, class, and documentation block into a unified knowledge graph. The graph consists of three node types: File Node, abstract syntax trees (AST) Node, and Text Node, as well as five relation types. This information captures the project’s file hierarchy, ASTs, and textual content, thus establishing a comprehensive semantic basis for downstream tasks. Further details on node design are provided in Part B and explained in Section 2.1.

The constructed knowledge graph is then utilized by PROMETHEUS’s multi-agent workflow (Part C), which involves issue classification, issue verification, context retrieval, patch generation, patch verification, and finally response generation. As an output, PROMETHEUS generates a validated patch to resolve the reported issue.

To our knowledge, PROMETHEUS distinguishes itself from existing graph-based approaches, such as RepoGraph [34], Alibaba’s Lingma Agent [31], and KGCompass [62], by constructing a unified, multi-modal knowledge graph that integrates repository structure, abstract syntax trees, documentation, and issue metadata. Unlike these methods, PROMETHEUS leverages modular, multi-agent workflows for context-aware retrieval, issue classification, and automated patch verification, enabling more precise reasoning, improved patch generation accuracy, and enhanced adaptability across diverse software engineering tasks.

### 2.1 Knowledge Graph Construction

To facilitate semantic understanding and retrieval across large-scale codebases, we propose a unified knowledge graph representation that integrates file structures, ASTs, and textual content into a coherent graph abstraction. Our knowledge graph is built around three core components: (1) defining a node and edge schema, (2) constructing the graph from source files, and (3) persisting the graph data in a Neo4j database.

**2.1.1 Graph Schema.** The knowledge graph represents codebases as heterogeneous graphs composed of three primary node types: ❶ `FileNode`, ❷ `ASTNode` and ❸ `TextNode` (shown in part B of Figure 1). A `FileNode` represents a file or directory with three attributes: a unique `node_id`, the `relative_path` from the repository root, and the basename of the file or directory. It anchors structural links in the knowledge graph. An `ASTNode` represents a Tree-sitter syntax node. It includes a unique `node_id`, the `start_line` and `end_line` indicating its position in the source file, the text of the code it covers (including comments), and its type, which specifies the Tree-sitter grammar node type. A `TextNode` represents a segment of text within the knowledge graph. It includes a unique `node_id`, associated `meta_data` describing the string, and the actual text content. This node type captures unstructured information from source files or documentation.

To capture relationships across `FileNodes`, `ASTNodes`, and `TextNodes`, we define five directed edge types. These relationships are

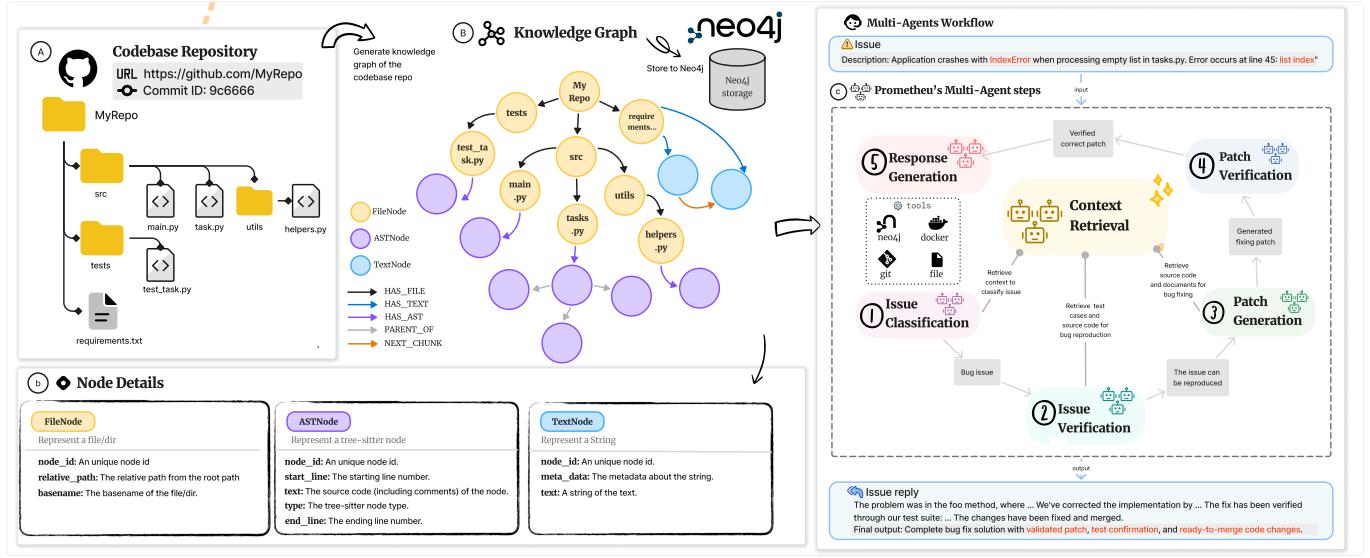


Figure 1: An overview of PROMETHEUS.

essential for representing the structural, syntactic, and lexical context necessary for code understanding and issue resolution. The HAS\_FILE edge (black) connects directories to their child files or sub-directories, preserving the repository hierarchy. The HAS\_AST edge (purple) links each file node to the root of its corresponding abstract syntax tree. PARENT\_OF edges (gray) connect AST nodes to reflect syntactic hierarchy within the tree. HAS\_TEXT edges (blue) associate file nodes with their segmented textual content, and NEXT\_CHUNK edges (orange) connect sequential text chunks to maintain document order. Together, these relationships enable the graph to represent structural, syntactic, and lexical information in an integrated and queryable format.

**2.1.2 Graph Construction and Neo4j Persistence .** Algorithm 1 outlines the construction of a codebase-level knowledge graph and its persistence to a Neo4j<sup>1</sup> database, which is a property graph database that stores nodes, relationships, and properties in a schema-flexible and query-efficient format.

It begins by initializing a node list  $\mathcal{N}$  and edge list  $\mathcal{E}$ , and setting a unique counter for node IDs. The root directory is wrapped as a FileNode and pushed onto a traversal stack (Lines 3–4). The algorithm then performs a depth-first traversal of the codebase (Line 5). For each directory, it creates a FileNode for each non-ignored child, adds a HAS\_FILE edge, and pushes the child onto the stack (Line 10–12).

If the file is a source file supported by Tree-sitter (Line 13), it is parsed into an abstract syntax tree (AST). Each AST node is added as an ASTNode, with PARENT\_OF edges capturing the tree structure. The AST root is connected to the file via a HAS\_AST edge (Line 16–18).

For Markdown or text files, the content is split into overlapping chunks. Each chunk becomes a TextNode (Line 19–22), linked

to the file with a HAS\_TEXT edge, and sequentially connected via NEXT\_CHUNK edges (Line 23–25) to preserve textual order. Finally, all nodes and edges (Line 27–29) are converted to Neo4j-compatible formats and written to the database, enabling structured querying and graph-based reasoning over the codebase.

## 2.2 Multi-agent Workflow

The PROMETHEUS multi-agent workflow (Part C of Figure 1) resolves Github issues through five coordinated agents. At its core is the Context Retrieval Agent, which supports three other agents by enabling precise access to relevant AST nodes, source files, and documentation from the Neo4j-backed graph.

Given an issue report, PROMETHEUS begins with ① *Issue Classification Agent*, where the agent categorizes the bug by querying the context retrieval agent. Once identified as a bug, ② *Issue Verification Agent* attempts to reproduce it using relevant test cases and code, supported by tools like Docker, Git, and direct file access. If the bug is reproducible, ③ *Patch Generation Agent* produces a fix by reasoning over the retrieved context. The ④ *Patch Verification Agent* then checks the patch's correctness by running tests. Finally, ⑤ *Response Generation Agent* produces a natural language summary, detailing the issue cause, applied fix, and test results.

**2.2.1 Tools for LLMs .** PROMETHEUS designs and implements four types of tools to assist LLMs.

**Neo4j tools:** Serves as the persistent backend for the knowledge graph. Nodes and edges representing files, AST structures, and documentation chunks are stored and queried using Cypher, which is Neo4j's declarative query language designed for expressing graph pattern matching. Agents interact with Neo4j through Cypher queries to retrieve relevant context.

**Docker tools:** Enable reproducible environments for code execution and testing. The Issue Verification Agent leverages Docker to

<sup>1</sup><https://neo4j.com>

**Algorithm 1:** Graph Construction and Neo4j Persistence

---

**Input:** Root directory of codebase  $\mathcal{D}$   
**Output:** Persisted knowledge graph in Neo4j

- 1 Initialize empty node list  $\mathcal{N}$  and edge list  $\mathcal{E}$ ;
- 2 Set  $\text{node\_id} \leftarrow 0$ ;
- 3 Create `FileNode` for root directory and add to  $\mathcal{N}$ ;
- 4 Initialize stack  $S \leftarrow [\text{root directory}]$ ;
- 5 **while**  $S$  is not empty **do**
- 6     Pop  $(f, \text{parent})$  from  $S$ ;
- 7     **if**  $f$  is directory **then**
- 8         **foreach** child  $c$  in  $f$  **do**
- 9             **if** not ignored by `.gitignore` **then**
- 10                 Create `FileNode` for  $c$  with  $\text{node\_id} \leftarrow \text{node\_id} + 1$ ;
- 11                 Add node to  $\mathcal{N}$  and `HAS_FILE` edge to  $\mathcal{E}$ ;
- 12                 Push  $(c, \text{FileNode}_c)$  to  $S$ ;
- 13     **else if** Tree-sitter supports  $f$  **then**
- 14         Parse  $f$  into AST;
- 15         **foreach** AST node  $n$  (with depth limit) **do**
- 16             Create `ASTNode` with  $\text{node\_id} \leftarrow \text{node\_id} + 1$ ;
- 17             Add node to  $\mathcal{N}$  and `PARENT_OF` edges to  $\mathcal{E}$ ;
- 18             Add `HAS_AST` edge from `FileNode` to AST root;
- 19     **else if**  $f$  is Markdown or text **then**
- 20         Split  $f$  into overlapping text chunks;
- 21         **foreach** chunk  $t_i$  **do**
- 22             Create `TextNode` with  $\text{node\_id} \leftarrow \text{node\_id} + 1$ ;
- 23             Add node to  $\mathcal{N}$  and `HAS_TEXT` edge to  $\mathcal{E}$ ;
- 24             **if**  $i > 0$  **then**
- 25                 Add `NEXT_CHUNK` edge from  $t_{i-1}$  to  $t_i$  to  $\mathcal{E}$ ;
- 26     **foreach** node in  $\mathcal{N}$  **do**
- 27         Convert to Neo4j-compatible node and write to database;
- 28     **foreach** edge in  $\mathcal{E}$  **do**
- 29         Convert to Neo4j-compatible edge and write to database;

---

safely run the codebase and validate whether the reported issue can be reproduced in isolation.

*Git tools:* Provide access to the project's version history and commit metadata. They support context enrichment, such as identifying the commit that introduced the bug or retrieving related change logs, helping agents generate more accurate fixes.

*File tools:* Represent the raw file system of the target repository. During graph construction, files are parsed and encoded as `FileNode`, `ASTNode`, or `TextNode` instances depending on their type. These nodes are later used for retrieval and reasoning.

### 2.3 Context Retrieval Agent

At the core of the PROMETHEUS architecture is the Context Retrieval Agent, which serves three downstream agents: *Issue Classification Agent*, *Issue Reproduction Agent*, and *Patch Generation Agent* by

providing them with semantically relevant code and documentation snippets. Built as a modular LangGraph<sup>2</sup> subgraph, this agent performs a structured three-phase process: 1) retrieval, 2) selection, and 3) refinement. LangGraph is a framework for building stateful and multi-step workflows over language models.

In the retrieval phase, a user query is transformed into a contextual prompt and used to invoke Neo4j-backed structured tools that search a heterogeneous knowledge graph containing `FileNodes`, `ASTNodes`, and `TextNodes`. These tools support various lookup strategies (e.g., by filename, AST pattern, or documentation text). In the selection phase, the agent employs an LLM to filter and rank candidate snippets based on task relevance and token efficiency. If the selected context is insufficient determined by an LLM, the refinement phase triggers a follow-up query based on the LLM's assessment of missing information, repeating the cycle iteratively. This dynamic mechanism enables precise and focused retrieval of code context, thereby enhancing the reasoning capabilities of PROMETHEUS across diverse software engineering tasks.

The Context Retrieval Agent leverages a suite of structured tools built on top of Neo4j and Cypher queries, including file lookup, AST node search, documentation traversal, and code preview tools, to extract semantically rich and relevant context from the codebase. We give an example in Listing 1 that queries the knowledge graph for all child AST nodes within a class definition. It searches for nodes up to five levels deep under any `ExampleNode` node and returns their text and type information.

Listing 1: Find child AST nodes under `ExampleNode`

```
MATCH (parent:ASTNode type:'ExampleNode')-
[:PARENT_OF * 1..5] -> (child : ASTNode)
RETURN child.text, child.type
```

### 2.4 Issue Classification Agent

*Issue Classification Agent* serves as the entry point for GitHub issue processing in PROMETHEUS. It performs triage by analyzing the textual content of each issue and categorizing it into one of four types: *bug*, *feature*, *documentation*, or *question*. The agent retrieve information to analyze issue characteristics of the following information: 1) Error patterns suggesting bugs; 2) New functionality requests 3) Documentation gaps and 4) Knowledge-seeking patterns to search on files matching descriptions, similar existing features, documentation coverage and with Related Q&A patterns from the Neo4j-backed knowledge graph. Issues classified as *BUG* go to the next step for reproduction.

Listing 2: Prompt for issue classification

```
OBJECTIVE: Find ALL self-contained context needed to accurately classify this issue as a bug, feature request, documentation update, or question.
1. Analyze issue characteristics: Error patterns suggesting bugs; New functionality requests; Documentation gaps; Knowledge-seeking patterns.
2. Search strategy: Implementation files matching descriptions; Similar existing features; Documentation coverage; Related Q&A patterns.
```

### 2.5 Bug Reproduction Agent

The *Bug Reproduction Agent* verifies whether a reported bug can be reproduced. It begins by analyzing the issue description to generate test cases that are likely to trigger the reported failure. The agent automatically detects the project's build system (e.g., Maven, Gradle,

<sup>2</sup><https://www.langchain.com/langgraph>

npm) and integrates with common test frameworks to execute relevant test suites.

When standard patterns fall short, it uses LLM reasoning to synthesize custom build and test commands. These commands are then executed in isolated environments using Docker to ensure clean runs. The agent iteratively refines its reproduction strategy if the initial attempt fails, up to a defined retry limit. It parses logs and test outputs to identify failure modes, distinguishing between pre-existing and newly introduced issues.

Throughout this process, it records detailed reproduction metadata, including logs, commands, and affected files, which serve as evidence for downstream repair steps. This agent leverages tools such as Git for version inspection, Docker for isolated execution, and Neo4j-backed context retrieval to guide command generation and verification.

#### Listing 3: Prompt for issue reproduction

```
You are a bug fix verification agent. Your role is to verify whether a bug has been fixed by running the reproduction steps and reporting the results accurately.
Your tasks are to: 1. Execute the provided reproduction commands on the given bug reproduction file 2. If a command fails due to simple environment issues (like missing "/" prefix), make minimal adjustments to make it work 3. Report the exact output of the successful commands
Guidelines for command execution: - Start by running commands exactly as provided - If a command fails, you may make minimal adjustments like: * Adding "/" for executable files - Do NOT modify the core logic or parameters of the commands - Do NOT attempt to fix bugs or modify test logic
```

## 2.6 Patch Generation Agent

*Patch Generation Agent* is responsible for producing fixes for verified bugs. It begins by constructing a targeted retrieval query based on the issue title, description, and comments. This query instructs the system to gather relevant production code, such as function definitions, class declarations, and related module logic, while ignoring test files. With the retrieved context, the agent uses an LLM to reason about the root cause and propose edits directly in the source code. The resulting changes are applied to the codebase, and the agent then runs build and test commands to ensure that the fix resolves the bug without introducing regressions. A Git diff is generated to capture the final patch for inspection or submission.

The agent leverages several core tools during this process: Neo4j-backed context retrieval to locate semantically relevant code, Git operations to track edits, and Docker-based execution to validate the patch within isolated environments.

#### Listing 4: Prompt for patch generation

```
You are an expert software engineer specializing in bug analysis and fixes. Your role is to:
1. Carefully analyze reported software issues and bugs by: - Understanding issue descriptions and symptoms - Identifying affected code components - Tracing problematic execution paths
2. Determine root causes through systematic investigation: - Identify which specific code elements are responsible
3. Provide high-level fix suggestions by describing: - Which functions or code blocks need changes - Why these changes would resolve the issue
4. For patch failures, analyze by: - Identifying what went wrong with the previous attempt - Suggesting revised high-level changes that avoid errors
```

## 2.7 Patch Verification Agent

*Issue Verification Agent* is designed to validate whether a generated patch successfully resolves the reported bug. It reuses the reproduction file and commands associated with the original bug and executes them within a containerized environment. The agent follows a systematic process: it first attempts to execute the reproduction commands exactly as specified, but can apply minimal corrections for common execution issues, such as missing prefixes or path

adjustments. This execution is performed using the `run_command` tool, which operates within Docker to ensure isolation and reproducibility. The agent is guided by a system prompt that enforces strict constraints: it must not attempt to fix or modify the bug itself, and must only report exact execution outputs. Once the commands are run, the agent parses the test results using a structured LLM component to determine whether the bug has been fixed. A bug is considered resolved only if all tests pass without errors or failures. This agent uses LangChain's structured tools, container-based execution, and Git-based patch tracking to support reliable post-fix validation.

#### Listing 5: Prompt for patch verification

```
You are a bug fix verification agent. Your role is to verify whether a bug has been fixed by running the reproduction steps and reporting the results accurately.
Your tasks are to: 1. Execute the provided reproduction commands on the given bug reproduction file 2. If a command fails due to simple environment issues (like missing "/" prefix), make minimal adjustments to make it work 3. Report the exact output of the successful commands
Guidelines for command execution: - Start by running commands exactly as provided - If a command fails, you may make minimal adjustments like: * Adding "/" for executable files - Do NOT modify the core logic or parameters of the commands - Do NOT attempt to fix bugs or modify test logic
```

## 2.8 Issue Response Agent

*Issue Response Agent* is the final step in the PROMETHEUS workflow. Its job is to write a professional and user-friendly comment that will be posted back to the original GitHub issue. This response is based on several key inputs: the original issue content, the patch generated by the repair agent, and the results from the verification agent. The agent uses a language model to combine this information into a clear and concise message. It highlights the system's understanding of the bug, explains what changes were made to fix it, and confirms that the fix was successfully tested. The agent follows strict guidelines to keep the tone appropriate for open-source communication and avoids revealing that the message was generated automatically. It uses LangChain's message formatting and a structured prompt template to generate the final response, ensuring clarity and professionalism.

#### Listing 6: Prompt for issue response verification

```
You are the final agent in a multi-agent bug fixing system. Users report issues on GitHub/GitLab, and our system works to fix them. Your role is to compose the response that will be posted back to the issue thread. The information you receive is structured as follows:
1. Issue Information (from user): The original issue title, body, and any user comments 2. Edit agent response: Generated by our edit agent after editing the source code 3. Patch: Created by our fix generation agent to resolve the issue 4. Verification: Results from our testing agent confirming the fix works
Write a clear, professional response that will be posted directly as a comment. Your response should: - Be concise yet informative - Use a professional and friendly tone appropriate for open source communication - Reference the system's understanding of the issue (from Edit agent response) - Explain the implemented solution (from patch) - Include the successful verification results
```

## 3 Experimental Methodology

### 3.1 Research Questions

- **RQ1 (Effectiveness Comparison):** How effectively does PROMETHEUS resolve issues compared to related code agents on the SWE-bench benchmark?
- **RQ2 (Multilingual Support):** How effectively does PROMETHEUS resolve issues across multiple programming languages?
- **RQ3 (Real-world Issue Resolution):** How well does PROMETHEUS generalize to real-world issues?

- **RQ4 (Scalability):** To what extent is PROMETHEUS scalable and efficient in constructing and storing knowledge graph representations of real-world software projects?
- **RQ5 (Cost):** What is the API cost of using PROMETHEUS during evaluation?

### 3.2 Methodology for RQ1

We evaluate PROMETHEUS on **SWE-bench Lite**, a curated subset of 300 issue-commit pairs drawn from 11 of the 12 Python repositories in the full SWE-bench [23] benchmark. This subset preserves the distribution and difficulty of the original dataset while focusing on self-contained, functional bug fixes. SWE-bench Lite provides a practical balance between evaluation fidelity and computational cost, making it suitable for academic research with limited computational resources. Several related works have submitted results to the SWE-bench Lite leaderboard, offering a fair basis for comparison.

*Baselines.* We compare PROMETHEUS with several widely used code agents evaluated on SWE-bench Lite. The selection criterion is the availability of source code and issue patches, which excludes works that are not open-sourced [31]. We finally select AutoCodeRover [74], Agentless [54], RepoGraph [34], Aider [15], CodeR [8] and SWE-Fixer [55]. Our selection includes both academic systems (AutoCodeRover, Agentless, and RepoGraph) and industrial systems (Aider and CodeR from Huawei). RepoGraph and CodeR are further selected to represent approaches that incorporate graph-based codebase representations, which are closely related to the design of PROMETHEUS.

*Models.* We select DeepSeek-V3 model <sup>3</sup>for evaluation, which is chosen to balance performance and cost.

Since our method generates only one candidate patch per issue, we evaluate it using two metrics. The first is the plausible rate, defined as the percentage of generated patches that make the known failing tests pass [46]. The second is the pass@1 issue resolution rate, defined as the percentage of issues resolved on the first attempt, as verified by held-out tests. We compare the performance of PROMETHEUS with baseline results reported in their respective papers and on the SWE-bench leaderboard. In addition, we examine each individual repository to identify all valid patches and analyze the number of issues uniquely resolved by our models in comparison to the baselines.

### 3.3 Methodology for RQ2

SWE-bench Lite issues are limited to Python. To evaluate the effectiveness of PROMETHEUS in supporting multiple programming languages, we conduct experiments on **SWE-bench Multilingual**. SWE-bench Multilingual [64] is a newly introduced benchmark in the SWE-bench family, designed to assess the software engineering capabilities of LLMs across multiple programming languages. It comprises 300 curated tasks derived from real-world GitHub pull requests, spanning 42 repositories and 9 programming languages. The benchmark covers a diverse set of application domains, including web frameworks, data processing tools, core system utilities, and widely used libraries. To our knowledge, no prior work has evaluated on this benchmark, and ours is the first to do so.

<sup>3</sup><https://www.deepseek.com>

We use the DeepSeek-V3 model for evaluation. Since no existing methods have been evaluated on SWE-bench Multilingual, we do not perform direct comparisons. Instead, we analyze the strengths and weaknesses of our approach across the 9 supported programming languages, providing both a performance baseline and insights to guide future research.

To our knowledge, many of existing code agents, such as Agentless can only support the single programming language Python and our work is novel to be support multiple languages to contribute the generalization.

### 3.4 Methodology for RQ3

Beyond benchmark evaluations, we aim to assess the applicability of PROMETHEUS on open issues from real-world open-source projects. This setting is more complex than SWE-bench, which provides pre-configured Docker files for reproducing known failing tests. In contrast, open GitHub issues often lack reproduction setups. To address this, PROMETHEUS includes an *Issue Classification Agent* and a *Bug Reproduction Agent*, which identify bug characteristics and generate customized Docker environments for reproducing real-world issues. To our knowledge, such capability is not present in prior code agents, including AutoCodeRover [74], Agentless [54], LingmaAgent [31], and RepoGraph [34].

For this evaluation, we select three open-source projects based on the following criteria: (1) more than 10,000 stars and 2,000 forks, (2) at least 100 open issues, including labeled bug reports, and (3) active maintenance, defined as having commits within three days prior to data collection. We report the number of issues resolved by PROMETHEUS. For all successfully generated patches, we submit the corresponding solutions to the issue discussion threads.

### 3.5 Methodology for RQ4 & RQ5

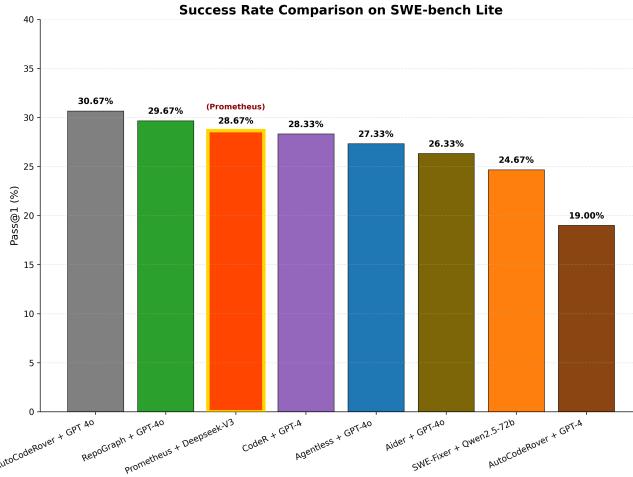
To evaluate the scalability and cost of PROMETHEUS, we analyze the knowledge graphs constructed for 11 Python repositories from the SWE-bench Lite benchmark. Each project is processed using PROMETHEUS to extract abstract syntax trees (ASTs), file structures, and textual content. These elements are stored as heterogeneous nodes and edges in a Neo4j database instance. In RQ5, in addition to scalability, we also report the API cost of evaluating SWE-bench Lite using the DeepSeek-V3 model.

## 4 Evaluation Results

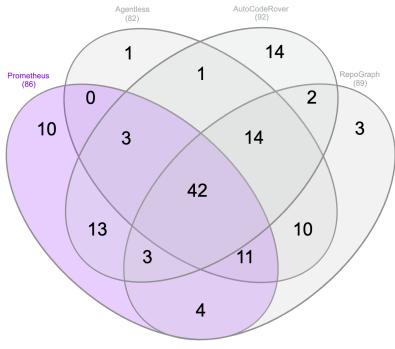
### 4.1 Answer to RQ1: Effectiveness Comparison

Across 300 instances in SWE-bench Lite, PROMETHEUS generated 229 plausible patches. Among these, 86 patches were verified as correct (i.e., the issue was resolved), while the remaining 143 were identified as overfitting cases, resulting in an overfitting rate of 62.4% (143/229). This result is consistent with prior findings [1, 52], which report that issue resolution is affected by a high rate of overfitting and highlight the need for further investigation.

Figure 2 presents a bar chart showing the pass@1 success rates of PROMETHEUS and the selected baselines on the SWE-bench Lite benchmark. Each bar represents the percentage of issues correctly resolved on the first attempt by a different system. PROMETHEUS, based on DeepSeek-V3, achieves a pass@1 success rate of 28.67%,



**Figure 2:** Pass@1 success rate comparison on the SWE-bench Lite benchmark



**Figure 3:** Venn diagram showing the complement and uniqueness of PROMETHEUS compared to three existing baselines: Agentless, AutoCodeRover, and RepoGraph.

outperforming Agentless with GPT-4o (27.00%) and several GPT-4-based baselines, including CodeR (28.33%) and AutoCodeRover with GPT-4 (19%), as well as SWE-Fixer (24.67%) based on finetune Qwen2.5-72b model.

PROMETHEUS performs slightly below AutoCodeRover with GPT-4o (30.67%) and RepoGraph built on the Agentless framework (29.7%). This difference can be attributed to the underlying model cost and capabilities. PROMETHEUS is implemented using DeepSeek-V3, a less costly model, whereas AutoCodeRover and RepoGraph rely on more advanced and expensive models such as GPT-4o. Due to budget constraints in academic settings, we do not utilize proprietary commercial models. GPT-4o charges \$10.00 per million output tokens, while DeepSeek-V3 ranges from \$0.28 to \$1.10, making DeepSeek-V3 approximately 9X to 36X cheaper.

Next, we compare the unique patches that are only generated by PROMETHEUS with DeepSeek-V3 model. Figure 3 shows a Venn diagram illustrating the overlap and unique issue resolutions of

**Table 1: Evaluation on Swe-bench Multilingual Benchmark**

Language	Issues	Resolved Instance	Resolved Rate
Java	43	15	34.9%
GO	42	6	14.3%
JS/TS	43	6	14.0%
C/C++	42	4	9.5%
Rust	43	4	9.3%
PHP	43	3	7.0%
Ruby	44	3	6.8%
Sum	300	41	13.7%

PROMETHEUS compared to three GPT-4o-based baselines: Agentless, AutoCodeRover, and RepoGraph. As shown in Figure 3, 42 issues are commonly resolved by all four approaches. Despite using a lower-cost model, PROMETHEUS uniquely resolves 10 issues that are not addressed by any of the other baselines. This demonstrates both the complementarity and distinct capability of PROMETHEUS in resolving GitHub issues.

Figure 4 presents a case study uniquely resolved by PROMETHEUS. The issue description is shown in Figure 4(a). The issue originates from the Django project, where the system check raises an E004 error when a sublanguage code (e.g., de-at) is used as LANGUAGE\_CODE, despite the base language (de) being listed in LANGUAGES. This violates Django's fallback mechanism, which allows sublanguage codes to default to their base language.

Figure 4(b) shows the Agentless patch, which incorrectly modifies LANG\_INFO based on the assumption that it influences validation, despite the actual check logic not referencing it. PROMETHEUS produces a correct patch in Figure 4(c) by identifying the faulty file and node through knowledge graph retrieval. The patch is semantically equivalent to the gold patch by SWE-bench shown in Figure 4(d).

*Answer to RQ1: PROMETHEUS resolves 28.7% (86/300) of issues in the SWE-bench Lite benchmark, outperforming four related approaches that rely on more costly models. In addition, PROMETHEUS uniquely resolves 10 issues not addressed by any of the three comparison benchmarks. These results demonstrate the effectiveness of PROMETHEUS.*

## 4.2 Answer to RQ2: Multilingual Support

Over 300 instances in SWE-bench Multilingual, PROMETHEUS generated 144 plausible patches and correctly resolved 41 issues, resulting in an overfitting rate of 71.5%. The details are given in Table 1.

Overall, PROMETHEUS achieves a 13.7% (41/300) issue resolution rate across seven programming languages, with the highest performance in Java (34.9%). The results show that PROMETHEUS resolved at least three issues in each language, with resolution rates ranging from 6.8% to 34.9%.

To our knowledge, PROMETHEUS is one of the few approaches that support multi-language issue resolution; others include SWE-agent [63] and CGM [49], which supports two programming languages: Python and Java. Most existing approaches are specific to Python [8, 34, 54, 62]. This is due to: (1) the use of the Tree-sitter parser,

Issue Description:		
<b>Issue Instance:</b> django_django-12286		
<b>Issue Title:</b> System check incorrectly raises error for sublanguage codes in LANGUAGE_CODE		
<b>Issue Description:</b> In Django 3.0.2, using a sublanguage code like de-at in LANGUAGE_CODE triggers a translation.E004 error, even when the base language de is in LANGUAGES. The check does not apply the documented fallback to the base language, resulting in a false-positive error.		
<b>(a) Issue Description</b>		
<p>An incorrect Patch Generated by Agentless:</p> <pre>The agentless patch is incorrect because LANG_INFO does not affect the system check logic.  --- a/django/conf/locale/__init__.py +++ b/django/conf/locale/__init__.py @@ -554,6 +554,9 @@ LANG_INFO = {     'vi': {         'name': 'Vietnamese',     }, +   'de-at': { +       'fallback': ['de'], +   }, }</pre>	<p>A correct patch generated by PROMETHEUS:</p> <pre>- a/django/core/checks/translation.py +++ b/django/core/checks/translation.py @@ -56,6 +56,10 @@ - if settings.LANGUAGE_CODE not in available_tags: -     return [E004] + + language_code = settings.LANGUAGE_CODE + + if language_code not in available_tags: + +     base_language = language_code.split('-')[0] + +     if base_language not in available_tags: + +         return [E004]</pre>	<p>A Gold Patch provided by SWE-Bench:</p> <pre>-- a/django/core/checks/translation.py +++ b/django/core/checks/translation.py @@ -55,7 +56,10 @@ - available_tags = i for i, _ in settings.LANGUAGES   'en-us' - if settings.LANGUAGE_CODE not in available_tags: + + try: +     get_supported_language_variant(settings.LANGUAGE_CODE) + + except LookupError: +     return [E004] + + else: -     return []</pre>
<b>(b) An Agentless Patch</b>	<b>(c) A PROMETHEUS patch</b>	<b>(d) A Gold patch provided by SWE-bench</b>

**Figure 4: A unique correct patch generated by PROMETHEUS in the SWE-bench Lite.**

which can parse multiple programming languages to construct a unified knowledge graph; and (2) the design of PROMETHEUS, which considers only general relationships rather than language-specific ones such as extends or implements. At the time of writing, our work is the first to appear on the leaderboard of SWE-bench Multilingual.

Most failure cases are caused by infinite loops during context retrieval in large codebases. Repositories in SWE-bench Multilingual (e.g., Java) are significantly larger than the Python projects in SWE-bench Lite, resulting in the generation of more file nodes, AST nodes, and text nodes, which increases the complexity of context retrieval.

*Answer to RQ2: PROMETHEUS resolves 13.7% (41/300) of issues in the SWE-bench Multilingual benchmark across 7 programming languages, using the DeepSeek-V3 model. The evaluation confirms that PROMETHEUS supports multi-language issue resolution.*

### 4.3 Answer to RQ3: Real-world Issue Resolution

To evaluate PROMETHEUS on real-world issues, we selected two representative open-source projects: LangChain [24] and Open Hands [2] that meet our selection criteria. We evaluated a total of eight GitHub issues across both repositories. Details are provided in Table 2. Among the eight issues, PROMETHEUS successfully generated patches for four. Each resolved issue had one generated patch, and all proposed solutions were submitted to the corresponding GitHub issue threads in the LangChain and Open Hands repositories.

We make two observations based on this evaluation. First, the results indicate that PROMETHEUS is applicable to real-world open issues, as it was able to generate at least one patch for each of the two projects. Second, PROMETHEUS is capable of handling diverse issue types. As shown in the third column of Table 2, the resolved issues

include schema generation, CLI iteration limits, Docker permission errors, and token input rate limits. The mean time to resolution for successful cases ranges from 25 to 48 minutes.

We also observe that PROMETHEUS fails in real-world open-source projects due to three main reasons. First, PROMETHEUS encounters API rate limits, which result in two failure cases. Second, the generated output does not always conform to the expected format, causing validation errors and one failure case. Third, PROMETHEUS is affected by context window limitations, which also lead to one failure case. These limitations suggest directions for future improvement of PROMETHEUS.

To our knowledge, compared to existing code agents such as AutoCodeRover and Agentless, our work is the first to evaluate on real-world open-source issues beyond the SWE-bench benchmark. This is enabled by the Issue Reproduction Agent in PROMETHEUS, which successfully creates reproduction environments for open issues without relying on pre-configured setups. The Issue Reproduction Agent depends on the Context Retrieval Agent for retrieving configuration information and assisting with infrastructure-level debugging.

Among the eight issues, PROMETHEUS successfully generated patches for four, resulting in a 50% success rate. Each resolved issue had one generated patch, and all proposed solutions were submitted to the corresponding GitHub issue threads in the LangChain and Open Hands repositories.

We make two observations based on this evaluation. First, the results indicate that PROMETHEUS is applicable to real-world open issues, as it was able to generate at least one patch for each of the two projects. Second, PROMETHEUS is capable of handling diverse issue types. As shown in the third column of Table 2, the resolved issues include schema generation, CLI iteration limits, Docker permission errors, and token input rate limits. The mean time to resolution for successful cases ranges from 25 to 48 minutes.

**Table 2: Real-world Issue Resolution Results.**

Project	Issue ID	Issue Type	Patch Generated
LangChain	#31808	Schema Generation	✓
	#31726	MLX Tool Calling	✗
	#31750	Azure OpenAI Validation	✗
	#32045	FAISS Similarity Search	✗
Open Hands	#9573	Configuration Field	✗
	#9426	CLI Iteration Limit	✓
	#9543	Docker Permissions	✓
	#9259	Token Input Rate Limit	✓

**Table 3: Graph Statistics of Projects in the SWE-Bench Lite.**

Project	File Nodes	AST Nodes	Text Nodes	HasAST	ParentOf
astropy	2131	223740	584	994	222746
django	9016	221661	949	2115	219546
flask	280	13134	123	78	13056
matplotlib	4558	158711	550	940	157771
pylint	2258	97301	642	1038	96263
pytest	653	47513	407	246	47267
requests	141	9185	26	75	9110
scikit-learn	1181	180707	277	680	180027
sphinx	2030	76874	462	634	76240
xarray	259	36522	73	153	36369
sympy	1896	633484	257	1268	632216

We also observe that PROMETHEUS fails in real-world open-source projects due to three main reasons. First, PROMETHEUS encounters API rate limits, which result in two failure cases. Second, the generated output does not always conform to the expected format, causing validation errors and one failure case. Third, PROMETHEUS is affected by context window limitations, which also lead to one failure case. These limitations suggest directions for future improvement of PROMETHEUS.

To our knowledge, compared to existing code agents such as AutoCodeRover and Agentless, our work is the first to evaluate on real-world open-source issues beyond the SWE-bench benchmark. This is enabled by the Issue Reproduction Agent in PROMETHEUS, which successfully creates reproduction environments for open issues without relying on pre-configured setups. The Issue Reproduction Agent depends on the Context Retrieval Agent for retrieving configuration information and assisting with infrastructure-level debugging.

*Answer to RQ3: PROMETHEUS successfully generated four out of eight patches for real-world issues in LangChain and Open Hands. To our knowledge, this is the first evaluation conducted on real-world issues beyond the SWE-bench benchmarks, thanks to the Reproduction Agent component, which creates container-based reproduction environments for real-world issues.*

#### 4.4 Answer to RQ4: Scalability

Table 3 reports the statistics of the knowledge graphs constructed by PROMETHEUS for 11 Python repositories from the SWE-bench Lite benchmark. For each project, the table presents the number of file nodes, AST nodes, and text nodes, as well as the number of HasAST and ParentOf edges. These statistics reflect the structure and scale of the extracted graphs across different repositories. For example,

**Table 4: API Cost of PROMETHEUS on DeepSeek-V3.**

Benchmarks	Issues	Total Cost	Cost/Issue
SWE-bench Lite	300	\$70.0	\$0.23
SWE-bench Multilingual	300	\$113.6	\$0.38

the Django repository contains 9,016 file nodes and 221,661 AST nodes, while the Sympy repository contains 1,896 file nodes and the highest number of AST nodes (633,484). The number of ParentOf edges correlates with the number of AST nodes and ranges from 9,110 (requests) to 632,216 (sympy). These measurements are used to assess the scalability of PROMETHEUS in constructing and storing code representations in Neo4j.

Neo4j Community Edition has been shown to reliably support several hundred million nodes and relationships, including over 100 million AST nodes, 10 million files, and 1 billion relationships. The graphs constructed from the 11 SWE-bench repositories represent only a small portion of the potential scale. Based on these results and Neo4j's documented capacity, PROMETHEUS is capable of persisting knowledge graphs with more than 100 million AST nodes and 10 million files, demonstrating its scalability for large-scale, real-world projects.

*Answer to RQ4: PROMETHEUS constructs and stores code knowledge graphs at a scale well within the capacity of Neo4j Community Edition. The evaluation on 11 SWE-bench repositories represents only a small portion of its potential, indicating that PROMETHEUS is scalable to large real-world projects.*

#### 4.5 RQ5: Monetary Cost

PROMETHEUS can be run on a machine with at least 36 GB of RAM, and the computational cost is manageable on a standard laptop. The majority of the cost comes from LLM API token usage. We report the monetary cost associated with using the DeepSeek-V3 API.

Table 4 reports the API cost of running PROMETHEUS on the DeepSeek-V3 model across two benchmarks: SWE-bench Lite and SWE-bench Multilingual. For each benchmark, 300 issue instances were evaluated. The total API cost for SWE-bench Lite is \$70.0, corresponding to a cost of \$0.23 per issue. For SWE-bench Multilingual, the total cost is \$113.6, with a cost of \$0.38 per issue. Compared to Agentless, which incurs a cost of \$0.70 per issue on SWE-bench Lite, PROMETHEUS reduces cost by approximately 67% (\$0.23 vs. \$0.70).

The higher cost for SWE-bench Multilingual is attributed to the inclusion of larger codebases written in languages such as C/C++, Java, and PHP, whereas SWE-bench Lite focuses only on Python. These larger codebases require more time for codebase navigation and context retrieval by PROMETHEUS. Nonetheless, the cost remains within a reasonable range, averaging \$0.38 per issue.

*Answer to RQ5: The average API cost per issue is \$0.23 for SWE-bench Lite and \$0.38 for SWE-bench Multilingual.*

## 5 Related Work

### 5.1 LLM-based Code Agents

The most closely related works are code agents for issue resolution beyond those compared in earlier sections. Now we discuss code agents related to program repair, test generation and fault localization. RepairAgent [5] uses an LLM-driven agent guided by a finite state machine for bug fixing; unlike our work, it assumes perfect fault localization and known failing tests. SpecRover [42] extracts code intents via multiple LLM agents with a focus on symbolic reasoning, which could complement future extensions of our system. SWE-Search [4] integrates Monte Carlo Tree Search for agent decision-making; in contrast, PROMETHEUS uses graph queries based on code structure.

LLM-based testing agents, such as ChatUniTest [11] and HalucinationConsensus [59], focus on test generation and validation, whereas PROMETHEUS targets repository-level issue resolution. AgentCoder [20] applies tree-of-thought prompting for small Python files; PROMETHEUS handles repository-scale, multi-language contexts with graph-based retrieval and build reasoning.

For fault localization, OrcaLoca [70] uses multi-agent simulations of human navigation but does not support issue reproduction or patch generation. LocAgent [12] employs a graph-guided fault localization strategy; PROMETHEUS extends this with classification, reproduction, and repair. AgentFL [38] simulates developer workflows with comprehension and dialogue agents for fault localization in Java; PROMETHEUS avoids document-guided search and supports multiple languages. LLM4FL [39] uses segmented test coverage and self-reflective LLM agents for Java fault localization; PROMETHEUS covers end-to-end issue resolution across languages.

### 5.2 Graph-Based Representation of Codebases

Modeling the codebase as a graph has emerged as an effective way to enable LLM reasoning. GraphCodeBert [18] integrates data flow graphs into code pre-training to improve representations for downstream tasks such as code search and summarization. Unlike PROMETHEUS, which constructs and queries a full knowledge graph of repository structure for issue resolution, GraphCodeBERT focuses on pre-training and does not support repository-level reasoning or agent-based repair. RepoGraph [34] constructs static program graphs from Python repositories, while PROMETHEUS is a more general design of graph nodes and relationship to support multilinguals. Liu et al. [29] proposed CodexGraph to retrieval interface that translates natural language queries into Cypher for answering code-related questions. Differently, PROMETHEUS integrates graph-based reasoning into a multi-agent pipeline for autonomous multi-language issue resolution. GraphCoder [28] builds a dependency graph among tokens for fine-grained code generation. Potentially, PROMETHEUS could include a dependency graph to improve the precision of context retrieval. KGCompass [62] constructs task-specific knowledge graphs to assist LLMs in software engineering tasks; PROMETHEUS, by contrast, builds general-purpose graphs from multiple modalities (AST, file, text) and supports broader downstream automation. GraphRAG [19] enhances retrieval-augmented generation by constructing and querying a domain knowledge graph to improve complex reasoning and retrieval precision. In contrast,

PROMETHEUS applies a similar graph retrieval mechanism specifically for automated, multi-language issue resolution, integrating the graph into a multi-agent pipeline for classification, reproduction, context retrieval, and patch generation.

### 5.3 Repository-level Context Retrieval

Recent work has explored repository-level context retrieval to support code understanding and repair. REPOFUSE [27] focuses on improving code completion by retrieving analogy- and rationale-based context within repositories, optimizing for latency and token efficiency. In contrast, PROMETHEUS targets constructing a heterogeneous knowledge graph and enabling end-to-end repair through multi-agent reasoning over repository-scale context. RepoCoder [73] implement retrieval mechanisms that go beyond file-level context by leveraging cross-file relations and iterative retrieval mechanisms. RepoCoder uses a similarity-based retriever and retrieves code chunks across files.

DocPrompting [75] retrieves documentation-relevant code snippets using embedding-based retrieval; PROMETHEUS instead uses structural graph traversal rather than retrieval-by-similarity for repository-scale code understanding. LTFix [13] encodes memory access patterns as typestates and retrieves related usage traces to support repair of memory errors in C codebases.

PROMETHEUS similarly retrieves context at repository scale but differs in its general-purpose, language-agnostic graph construction and retrieval model. Unlike prior work that targets specific tasks such as code completion in RepoCode or memory errors in LTFix, PROMETHEUS supports issue resolution across diverse tasks and programming languages. PROMETHEUS also supports user-driven Cypher queries over persisted Neo4j graphs, enabling scalable, structured retrieval beyond pre-defined retrieval logic.

CodeRAG-Bench [53] evaluates whether retrieval-augmented generation improves code generation across tasks including repository-level problems. In contrast, PROMETHEUS applies retrieval-augmented techniques specifically for automated issue resolution, constructing and querying a multi-language knowledge graph as part of a multi-agent pipeline that covers classification, reproduction, and patching.

### 5.4 Automated Program Repair

Issue resolution is a subset of automated program repair (APR), which has developed into three major categories over the past decade: search-based, semantic-based, and learning-based repair techniques [17].

Learning-based approaches, such as Codit [6], Modit [7], CURE [22], DLFix [26], RewardRepair [68], SelfAPR [67], and Tufano's work [50, 51], typically rely on supervised learning over past commits, training models to translate buggy code into correct code. Some recent work has explored self-supervised learning to avoid the need for commit data [3, 65].

Search-based approaches, such as GenProg [25], Relifix [48], ssFix [57], and others [9, 16, 32, 43, 56, 66, 71], define a space of edit patterns and use search algorithms to identify plausible patches.

Semantic-based approaches, including SemFix [33], Nopol [60], and others [14, 44, 45, 58], formulate repair as a constraint satisfaction problem and synthesize patches that meet those constraints.

## 6 Threats to Validity

*Internal Validity.* The system may be affected by implementation bugs or configuration issues. LLM outputs are non-deterministic, which can lead to variation across runs. To mitigate this, we have open-sourced our implementation for future examination.

*External Validity.* The evaluation is based on SWE-bench Lite, SWE-bench Multilingual, and a few real-world repositories. Results may not generalize to all codebases or environments. To support broader evaluation, we provide tools to run the system on arbitrary real-world issues.

*Construct Validity.* Success is defined as passing held-out test cases specified by SWE-bench, which may not capture all valid fixes. Some plausible patches may be incorrectly marked as failures. To address this, we manually inspect PROMETHEUS generated patches to compare them with gold patches.

## 7 Conclusion

We introduced PROMETHEUS, a multi-agent system that constructs a unified knowledge graph to support multi-language issue resolution. It operates on arbitrary GitHub repositories without relying on pre-built environments, incorporating agents for issue classification and reproduction. PROMETHEUS resolves 28.67% of issues in SWE-bench Lite and 13.7% in SWE-bench Multilingual, covering seven languages. It is the first system evaluated on SWE-bench Multilingual and achieves lower cost compared to GPT-4o-based baselines. PROMETHEUS also resolves real-world issues in public repositories, demonstrating applicability beyond benchmarks.

## References

- [1] Reem Aleithan, Haoran Xue, Mohammad Mahjor, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992 [cs.SE] <https://arxiv.org/abs/2410.06992>
- [2] All-Hands-AI Team. 2024. OpenHands: An Open Platform for AI Software Developers. <https://github.com/All-Hands-AI/OpenHands>. Accessed: 2025-07-17.
- [3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems*.
- [4] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2025. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. arXiv:2410.20285 [cs.AI] <https://arxiv.org/abs/2410.20285>
- [5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2188–2200. <https://doi.org/10.1109/ICSE55347.2025.00157>
- [6] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* (2020). <https://doi.org/10.1109/TSE.2020.3020502>
- [7] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [8] Dong Chen, Shaolin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheschkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304 [cs.CL]
- [9] L. Chen, Y. Pei, and C. A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [10] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [11] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. arXiv:2305.04764 [cs.SE] <https://arxiv.org/abs/2305.04764>
- [12] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. arXiv:2503.09089 [cs.SE] <https://arxiv.org/abs/2503.09089>
- [13] Xiao Cheng, Zhihao Guo, Huan Huo, and Yulei Sui. 2025. Tracing Errors, Constructing Fixes: Repository-Level Memory Error Repair via Typestate-Guided Context Retrieval. arXiv:2506.18394 [cs.SE] <https://arxiv.org/abs/2506.18394>
- [14] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-Avoiding Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 8–18. <https://doi.org/10.1145/3293882.3330558>
- [15] Paul Gauthie. 2024. Aider is ai pair programming in your terminal. <https://aider.chat>
- [16] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 19–30. <https://doi.org/10.1145/3293882.3330559>
- [17] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
- [18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=jLoC4ez43PZ>
- [19] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A. Rossi, Subhabrata Mukherjee, Xianfeng Tang, Qi He, Zhigang Hua, Bo Long, Tong Zhao, Neil Shah, Amin Javari, Yinglong Xia, and Jiliang Tang. 2025. Retrieval-Augmented Generation with Graphs (GraphRAG). arXiv:2501.00309 [cs.IR] <https://arxiv.org/abs/2501.00309>
- [20] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010 [cs.CL] <https://arxiv.org/abs/2312.13010>
- [21] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOT: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery.
- [22] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*.
- [23] Carlos E Jimenez, John Yang, Alexander Wetzig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQM66>
- [24] LangChain Development Team. 2022. LangChain: Building applications with LLMs through composability. <https://github.com/langchain-ai/langchain>. Accessed: 2025-07-17.
- [25] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 54–72. <https://doi.org/10.1109/TSE.2011.104>
- [26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 602–614. <https://doi.org/10.1145/3377811.3380345>
- [27] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, wei jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. arXiv:2402.14323 [cs.SE] <https://arxiv.org/abs/2402.14323>
- [28] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Code Context Graph-based Retrieval and Language Model. arXiv:2406.07003 [cs.SE] <https://arxiv.org/abs/2406.07003>
- [29] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 142–160. <https://doi.org/10.18653/v1/2025.nacl-long.7>
- [30] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtiar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraein Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii

- Osae Osa Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muenennighoff, Xiangru Tang, Muh-tasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostafa Pathwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] <https://arxiv.org/abs/2402.19173>
- [31] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhu Li, Fei Huang, and Yongbin Li. 2025. Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration. arXiv:2406.01422 [cs.SE] <https://arxiv.org/abs/2406.01422>
- [32] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*.
- [33] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. <https://doi.org/10.1109/ICSE.2013.6606623>
- [34] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. arXiv:2410.14684 [cs.SE] <https://arxiv.org/abs/2410.14684>
- [35] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 42 (Jan. 2025), 28 pages. <https://doi.org/10.1145/3697010>
- [36] Anvith Pabba, Alex Mathai, Anindya Chakraborty, and Baishakhi Ray. 2025. SemAgent: A Semantics Aware Program Repair Agent. arXiv:2506.16650 [cs.SE] <https://arxiv.org/abs/2506.16650>
- [37] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. CW-Eval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. 33–40. <https://doi.org/10.1109/LLM4Code66737.2025.00009>
- [38] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixia Wang, Xiaoliang Li, and Xiaoguang Mao. 2025. AgentFL: Scaling LLM-based Fault Localization to Project-Level Context. arXiv:2403.16362 [cs.SE] <https://arxiv.org/abs/2403.16362>
- [39] Md Nakhlra Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2025. A Multi-Agent Approach to Fault Localization via Graph-Based Retrieval and Reflexion. arXiv:2409.13642 [cs.SE] <https://arxiv.org/abs/2409.13642>
- [40] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2025. Agentic AI Software Engineers: Programming with Trust. arXiv:2502.13767 [cs.SE] <https://arxiv.org/abs/2502.13767>
- [41] Baptiste Rozière, Jonas Gehring, Fabian Glocleck, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémie Rapin, Artiom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhao Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] <https://arxiv.org/abs/2308.12950>
- [42] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. arXiv:2408.02232 <https://arxiv.org/abs/2408.02232>
- [43] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE '19)*. IEEE Press, 13–24. <https://doi.org/10.1109/ICSE.2019.00020>
- [44] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [45] Ridwan Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 1–36.
- [46] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (Bergamo, Italy) (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. <https://doi.org/10.1145/2786805.2786825>
- [47] Marc Szafraniec, Baptiste Rozière, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code translation with Compiler Representations. *ICLR* (2023).
- [48] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 471–482. <https://doi.org/10.1109/ICSE.2015.65>
- [49] Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Lin-chao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. 2025. Code Graph Model (CGM): A Graph-Integrated Large Language Model for Repository-Level Software Engineering Tasks. arXiv:2505.16901 [cs.SE] <https://arxiv.org/abs/2505.16901>
- [50] Michele Tufano, Jevgenija Pantichina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE '19)*. IEEE Press, 25–36. <https://doi.org/10.1109/ICSE.2019.00021>
- [51] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages. <https://doi.org/10.1145/3340544>
- [52] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-Bench Really Solved Correctly? An Empirical Study. arXiv:2503.15223 [cs.SE] <https://arxiv.org/abs/2503.15223>
- [53] Zora Z. Wang, Akari Asai, Xinyan V. Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can Retrieval Augment Code Generation? (2024).
- [54] Chunqiu Steven Xie, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. <https://doi.org/10.1145/3715754>
- [55] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. *arXiv preprint arXiv:2501.05040* (2025).
- [56] Qi Xin and Steven Reiss. 2019. Better Code Search and Reuse for Better Program Repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. 10–17. <https://doi.org/10.1109/GI2019.00012>
- [57] Q. Xin and S. P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [58] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (Buenos Aires, Argentina) (ICSE '17)*. IEEE Press, 416–426. <https://doi.org/10.1109/ICSE.2017.45>
- [59] Qinghua Xu, Guancheng Wang, Lionel Briand, and Kui Liu. 2025. Hallucination to Consensus: Multi-Agent LLMs for End-to-End Test Generation with Accurate Oracles. arXiv:2506.02943 [cs.SE] <https://arxiv.org/abs/2506.02943>
- [60] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lameiras, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).
- [61] Aidan Z. H. Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. 2025. Enhancing Repository-Level Software Repair via Repository-Aware Knowledge Graphs. *arXiv preprint arXiv:2503.21710* (2025).
- [62] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. <https://arxiv.org/abs/2405.15793>
- [63] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. <https://arxiv.org/abs/2405.15793>
- [64] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. arXiv:2504.21798 [cs.SE] <https://arxiv.org/abs/2504.21798>
- [65] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *International Conference on Machine Learning (ICML)*.
- [66] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825. <https://doi.org/10.1016/j.jss.2020.110825>
- [67] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfFAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (Rochester, MI, USA) (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages. <https://doi.org/10.1145/3551349.3556926>
- [68] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. <https://doi.org/10.1145/3510003.3510222>
- [69] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. <https://doi.org/10.1145/3510003.3510222>

- 3597503.3623337
- [70] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. OrcaLoca: An LLM Agent Framework for Software Issue Localization. arXiv:2502.00350 [cs.SE] <https://arxiv.org/abs/2502.00350>
  - [71] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. In *IEEE Transactions on Software Engineering*.
  - [72] Andrew Zakanov. [n. d.]. Meet Junie, Your Coding Agent by JetBrains. <https://blog.jetbrains.com/junie/2025/01/meet-junie-your-coding-agent-by-jetbrains/>. Accessed: 2025-06-17.
  - [73] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2471–2484. <https://doi.org/10.18653/v1/2023.emnlp-main.151>
  - [74] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1592–1604. <https://doi.org/10.1145/3650212.3680384>
  - [75] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating Code by Retrieving the Docs. arXiv:2207.05987 [cs.CL] <https://arxiv.org/abs/2207.05987>